Composition with Guarantees for High-integrity Embedded Software Components Assembly

Project Number 216682

D5.1 – Technology-neutral specification of property-preserving run-time environment

Version 1.0

31 January 2011

Final

Public Distribution

INT, UPD, CNR-ISTI, TCF, ATEGO, INRIA, FhG, AICAS, TOG, EAB, ENEA, MDH, ATOS

Project Partners: **Aicas, Atego, Atos Origin, CNRI-ISTI, Enea, Ericsson, Fraunhofer, FZI, GMV Aerospace & Defence, INRIA, Intecs, Italcertifier, Maelardalens University, Thales Alenia Space, Thales Communications, The Open Group, University of Padova , University Polytechnic of Madrid**

Contents

Document Control

Version	Status	Date
0.1	proposal for deliverable structure	2010-01-24
0.2	initial inputs for Section 3 and introduction	2010-02-17
0.3	first draft for Section 2.1	2010-03-15
0.4	minor restructuring; added some content to Section 2 based on the meetings in Bruxelles and Stockholm	2010-06-16
0.5	restructuring following e-mail discussions	2010-09-02
0.6	version before teleconference with additional contributions by some partners	2010-09-09
0.7	version after telco, prepared for the upcoming work	2010-09-10
0.8	integrate requirements for schedulability analysis from D4.3	2010-09-19
0.9	integrate comments from space domain; integrate requirements for predictable memory consumption (from D4.3)	2010-09-26
0.91	integrate results from discussion at Karlsruhe meeting	2010-10-04
0.92	completed sections on feasibility of predictability requirements; added dependability requirements	2010-11-29
0.93	integrated improvements of Ada section from Atego; feasibility of dependability requirements for Java	2010-12-08
0.94	integrated comments from Barcelona meeting and updated sections on dependability requirements	2010-12-22
0.95	feasibility of dependability requirements on target REs	2011-01-17
0.96	version for internal review	2011-01-21
1.0	corrections after internal review	2011-01-30

This page was intentionally left blank.

Executive Summary

This document collects requirements on run-time environments in order to ensure consistency between model-based static analyses and run-time environments. The requirements reflect assumptions that analysis models make on run-time environments. Run-time mechanisms, such as run-time monitoring and enforcement, that complement model-based static analyses are identified. The requirements are evaluated by end users in order to ensure that they are compatible with end user needs. Their feasibility on the targeted run-time environments — Ada Real-time, Java/RTSJ, Real-time Linux and OSE — is evaluated.

1 Introduction

Modern development methodologies seek and promote as early verification as possible of the extra-functional *properties* that must be exhibited by the system. The notion of *property* must be understood as follows: when extra-functional attributes used as input for some form of system analysis confirm required qualities of the system, then they become "properties" of the system. In practise, they become *constraints* on the implementation and execution of the system, as they represent the "feasibility space" within which the system or a component of it does actually correspond to the stipulations made during the analysis. It therefore follows that any violation of those properties occurring at run-time may invalidate the analysis results and cause the system to deviate from its expected behaviour as predicted by the analysis.

For this reason, the design methodology adopted in CHESS supports *property preservation*, to ensure that those extra-functional properties are effectively conveyed into the implementation and monitored/enforced at run-time.

The run-time environment is thus required to provide adequate mechanisms to realise property preservation. It is the goal of this document to identify those mechanisms, so as to ensure conformance of run-time environments with the model-based approach developed in CHESS. This document is the starting point for WP5 and serves as a guide to the development of run-time environments and model-to-code transformations that will be carried out in this work package.

1.1 Recapitulation of the CHESS approach

CHESS has adopted a component-oriented approach. The *component model* provides the conceptual and technical means to specify the software system as an assembly of components.

The *computational model* with which the component model is bound during system production is the scientific means that permits to warrant static analysability of the system in the time, space and communication dimensions. The computational model encompasses all the semantic assumptions and constraints required by its related analysis techniques.

The *programming model* for a target implementation is the means to safely reject all language constructs that do not conform with the synchronisation and execution semantics permitted by the analysis framework and that can thus undermine the analysis assumptions.

An *execution platform* that fits our concept hosts and executes software entities and is in charge of actively preserving the software and system properties asserted during static analysis, those which cannot be ensured by static (i.e., software production) means.

Figure 1 recapitulates the relationship between the component, computational and programming model and the underlying execution platform. In D2.1, we explained in detail these concepts and the relationship between the component and computational model.

We said earlier that what we term "properties" in our approach are extra-functional attributes set on the user model, which are used as input values for static analysis of the system. The preservation of those properties (the attributes which resulted in an accepted system) can be considered as a requirement imposed on the implementation and execution technology to preserve the validity of the analysis results; this is achieved by ensuring that the software at run-time conforms at all times with its analysed model.

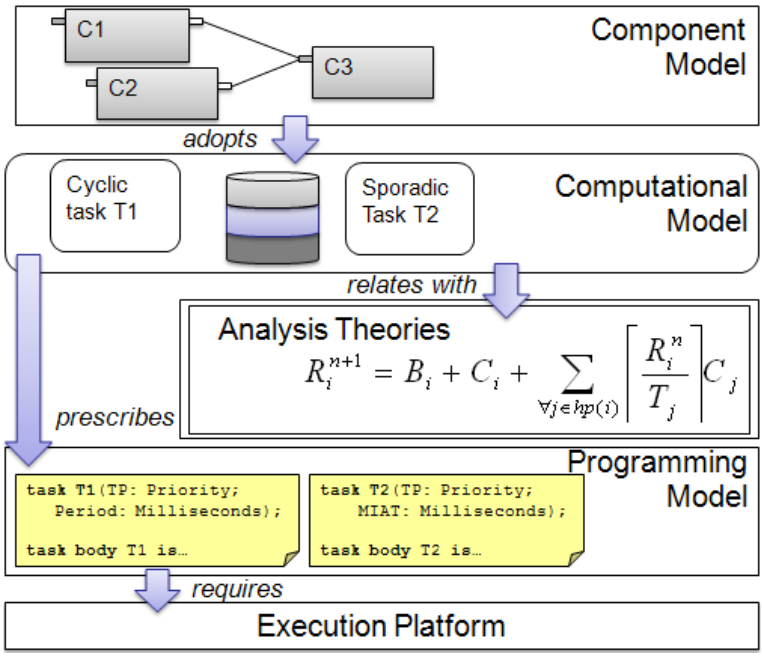


Figure 1: Overview of the relation between the Component, Computational and Programming Model and the underlying execution platform.

While the programming model is used as a *static* assurance of compliance, property preservation is carried out with the deployment of mechanisms (typically ratified in suited code archetypes) that operate at run-time, for the properties of interests that depend on software execution.

One of the most important goals for CHESS is the realisation of a component model that is neutral with respect to the computational models adopted by the CHESS target domains. The component model includes the means to annotate the design entities with the necessary extra-functional attributes. The sought neutrality of the component model however requires that the semantics of the extra-functional attributes is not fixed at the component-model level but it is left open to interpretation. The precise binding of those attributes to a given semantics occurs when they are used in the model representation used as input for the analysis. This analysis model conforms with an analysis meta-model suited for the adopted computational model of interest.

This situation is indeed good news, as it implies that the component model, for what concerns the time, space and communication dimensions, does not generate any requirements that need property preservation. All those requirements instead originated from the adopted analysis theories, hence from the computational model, as reflected in its analysis meta-model. Section 2 of this document elaborates on the requirements related to dependability, predictability, isolation and transparency that must be fulfilled by the run-time environments demonstrated in CHESS (via techniques that may include for example, partitioning, communication transparency, distribution transparency, etc...).

Two different scenarios may arise in scoping the relation between the computational models relevant to the different industrial domains of interest to CHESS.

In the first and more favourable scenario (see Figure 2), information relevant for analysis is extracted from the CHESS design model and attributed to an analysis model that conforms to a meta-model that is common to all application domains addressed in CHESS. This common analysis meta-model is feasible if we are able to: (i) factorise the analysis needs that stem from the computational mod-

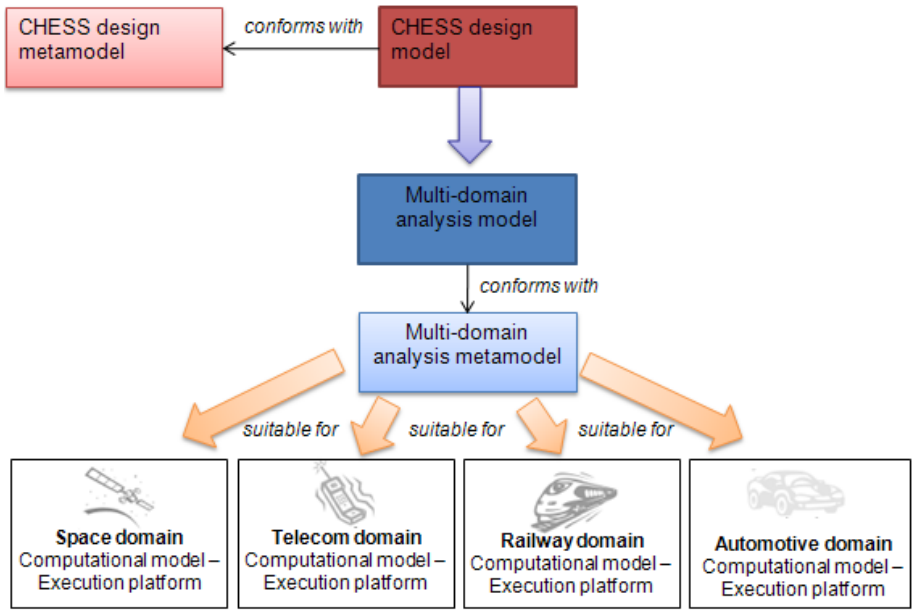


Figure 2: Overview of the approach with a common multi-domain meta-model.

els of all the domains, (ii) provide a non-ambiguous semantics for all the analysis concepts and attributes. In particular, careful attention should be paid to understanding whether the interpretation of certain concepts in different domains is contradictory or incompatible. Since requirements on property preservation originate from the analysis needs, it would in this case be possible to achieve a factorisation of the requirements with a bearing on property preservation imposed on the execution platforms of interest.

In the alternative scenario, each domain retains a distinct analysis meta-model (see Figure 3). This situation will occur if it is not possible to achieve a factorisation of the analysis needs of the various domain; each single domain retains disjointly its analysis techniques and therefore also the definition of requirements that need property preservation have to be performed for each single domain in isolation.

An intermediate scenario is also possible, depending on the extent to which the progress of WP4 may be able to factorise the analysis needs across all industrial domains. In that scenario, there would be a sort of core analysis concepts, common to all domains, complemented by a set of domain-specific ones. In that case there would be a set of requirements for property preservation common to all domains, and additionally a set of property preservation requirements specific to each domain. Requirements specific to a single domain would be fulfilled only by the execution platform of choice of the domain.

1.2 Outline

Section 2 collects and discusses requirements on run-time environments. Section 3 introduces the targeted run-time environments and discusses if and how the requirements from Section 2 are supported in these environments. Section 4 concludes the report.

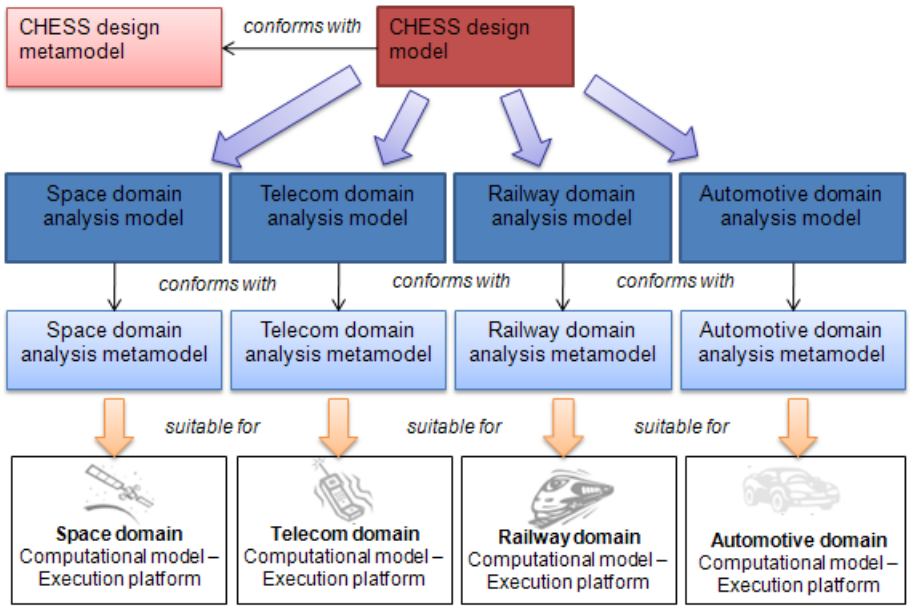


Figure 3: Overview of the approach with a distinct analysis meta-model (hence analysis model) per target domain.

2 Collection and Evaluation of Requirements

Outline Following a model-driven approach, requirements on the run-time environments originate from assumptions made by analysis models. Task 3.4 (reported in D3.3) and Task 4.4 (reported in D4.3) describe such assumptions and identify required features of run-time environments. Sections 2.1, 2.2 and 2.3 of this deliverable recapitulate these requirements. The requirements are presented in a narrative style, explaining the origin and reason for each requirement in the accompanying text. Each requirement is formulated in a technology-neutral way, avoiding terminology and concepts that are specific to certain target run-time environments. Section 2.4 proposes additional requirements for run-time monitoring and run-time property enforcement that come directly from end users and serve to complement static analysis. Section 2.5 evaluates the requirements from the end users' point of view. If certain requirements prove too restrictive for certain target domains, we look for ways to relax critical assumptions through run-time property enforcement. To provide an overview, Section 2.6 summarises the requirement list in tabular form.

Requirements on run-time environments vs. requirements on model-to-code transformers We note that there are two kinds of requirements on run-time environments: the first kind requires that run-time environments *provide desired features*; the second kind requires that run-time environments *do not provide undesired features that are in the way of analysability*. Examples of the first kind of requirement are the need for high resolution timers, the need for a priority-based scheduler, or the need for cost monitoring and cost enforcement. An example for the second kind of requirement is the need to forbid dynamic thread creation for the sake of analysability. The second kind of requirement can either be achieved by using a programming language that supports a restricted programming model (e.g., a language that does not provide an instruction for dynamic thread creation), or by designing model-to-code transformers to avoid undesired run-time features (e.g., model-to-code transformers should never generate instructions for dynamic thread creation). Thus, the second kind of require-

ment can be interpreted as a requirement on model-to-code transformers, rather than a requirement on run-time environments. The following collection includes both kinds of requirements.

2.1 Requirements for Schedulability Analysis

Schedulability analysis assumes that applications can be decomposed into a number of separate *tasks*, each performing a certain *job* within a single thread of control. Each thread of control spends part of its time waiting for the next release to perform its job. Job releases can either be time-triggered or event-triggered. The schedulability analysis methods considered in CHESS can handle at least two job release patterns: *periodic job releases* and *sporadic job releases*. An important parameter of both these release patterns is the time interval that is allowed to pass in between subsequent releases. In the periodic case, this interval represents a *period*: jobs must be released at the beginning of each period. In the sporadic case, the interval represents a *minimal inter-arrival time (MIAT)*: the time between two job releases must not be smaller than the MIAT. The run-time environments must enforce driftless and jitterless periods and MIATs for periodic and sporadic job releases respectively. Thus, we derive the following requirements on the run-time environments:

(Req 1) Run-time environments must provide means to implement periodic and sporadic tasks. Both program-driven and interrupt-driven sporadic tasks must be supported.
(Req 2) Run-time environments must enforce periodic job releases for periodic tasks.
(Req 3) Run-time environments must enforce MIATs between job releases for sporadic tasks.

The *response time* is the the time between a job release and the job completion. Schedulability analysis attributes a *deadline* to each job (on a per-task basis). It is the goal of schedulability analysis to statically predict that a given set of tasks can be scheduled in a manner such that all jobs meet all of their timing requirements. This is summarised in Figure 4.

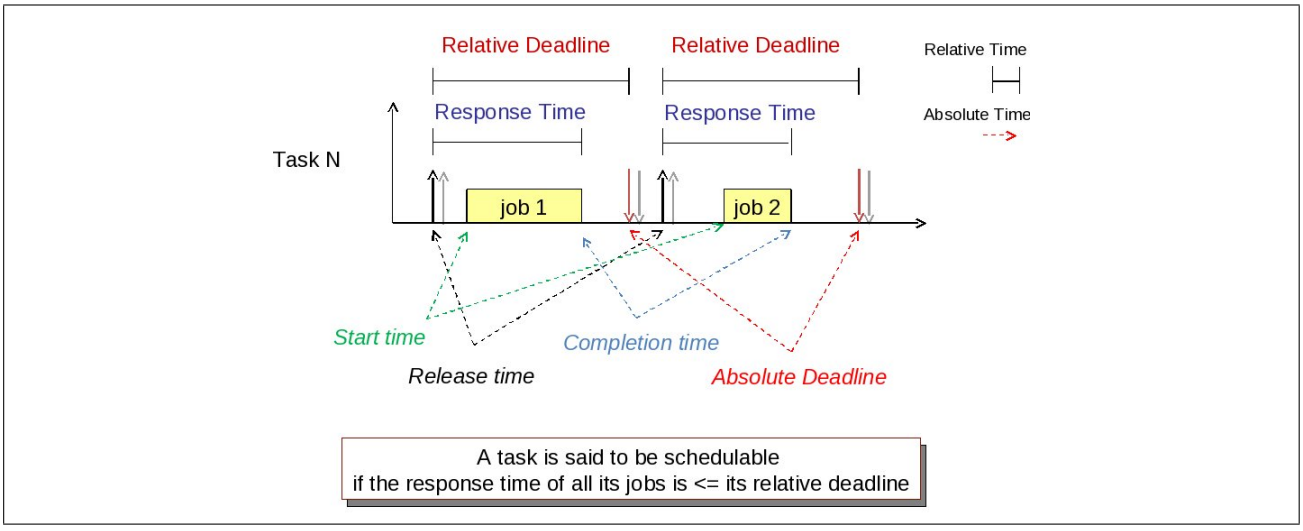


Figure 4: Schedulability analysis: task execution and timing requirements.

Figure 5 shows how a system of tasks is represented abstractly. Such a system consists of a fixed number of tasks, each of which has four parameters, as shown in Figure 5. Note that the number of tasks is fixed for each task system. This imposes the following requirement on run-time environments:

- N: number of tasks
- For each i in $\{1, \ldots, N\}$, the i-th task is represented by $\tau_i(\phi_i, T_i, C_i, D_i)$:
  - ϕ_i: phase (release time of the first job of task i)
  - T_i: period or minimum inter-arrival time (MIAT)
  - C_i: worst-case execution time (WCET), the longest execution time among all possible jobs of a task
  - D_i: relative deadline

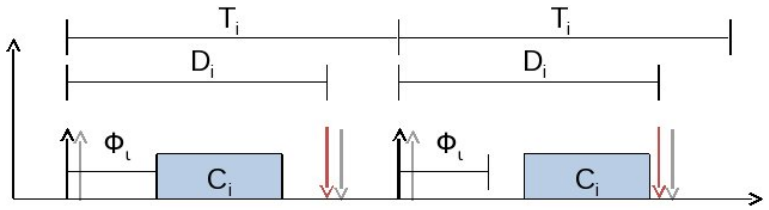


Figure 5: Schedulability analysis: abstract representation of task systems.

Response Time: $R_i = C_i + I_i + B_i + Ins_i$

- *Execution time* C_i: CPU time spent executing the task
- *Scheduling interference* I_i: CPU time spent executing higher-priority tasks
- *Blocking time* B_i: CPU time spent executing lower-priority tasks due to shared resource contention
- *Interference by non-task entities and run-time mechanisms* Ins_i: Hardware clock ISR, external interrupts, overhead of the kernel/RTOS, etc.

Figure 6: Schedulability analysis: computing response times.

(Req 4) Run-time environments must provide means to implement systems where all threads (i.e., the run-time implementations of tasks) are created in a system initialisation phase, and no more threads are dynamically created thereafter.

(Req 5) Run-time environments must provide means to implement systems where all threads do not freely terminate.

Schedulability analysis relies on *response time analysis* for calculating upper bounds on response times. Response times are calculated as the sum of execution times, scheduling interference, blocking times and interference by non-task entities and run-time mechanisms, as shown in Figure 6. The run-time environments need to be designed so that bounds on these quantities can be predicted statically.

In order to facilitate prediction of *execution times*, it is important that each job is associated with a worst-case execution time.

(Req 6) Run-time environments must ensure that each job has an associated worst-case execution time.

More generally, it is important that, for each job, it is clear which resources are needed for executing it:

(Req 7) Run-time environments must ensure that, for each job, the resources that are needed for executing the instruction are predictable (e.g., CPU, network and communication resources, thread-shared memory).

Finally, it is important that execution times for behind-the-scene management tasks (e.g., automatic memory management) are predictable. For instance, the run-time environment should not rely on an automatic garbage collector, except perhaps a real-time garbage collector.

(Req 8) Run-time environments must ensure that all management tasks that operate underneath the application software (e.g., automatic garbage collection) are time-bounded.

In order to facilitate prediction of *scheduling interference*, the run-time environment must provide a scheduling algorithm that scheduling analysis can handle. The scheduling analysis methods considered in CHESS support fixed-priority preemptive scheduling and earliest-deadline-first scheduling. The end user partners require that at least fixed-priority preemptive scheduling is supported.

(Req 9) Run-time environments must support fixed-priority preemptive scheduling. The association between threads and priorities must be static (except for dynamic priority changes due to priority inheritance).

(Req 10) Run-time environments may optionally support earliest-deadline-first scheduling.

In most applications, tasks are not independent of each other, but need to communicate. Schedulability analysis assumes that communication between concurrent tasks is carried out via shared memory. In order to ensure mutually exclusive memory access, it is required that all thread communication is achieved through *monitors* [7]. Monitors associate a piece of shared memory with a set of procedures, and the run-time system ensures that no two calls to procedures of the same monitor execute concurrently. Typically, monitors are implemented in terms of locks, where a lock is acquired upon entry to a monitor procedure and released upon procedure exit. Monitor synchronisation has the property that a lock is always released by the same thread that has acquired it. This property is crucial in order to define meaningful protocols for preventing priority inversion. Both Ada's protected objects and Java's synchronised methods are based on monitors.

(Req 11) Run-time environments must support monitors as a mechanism for synchronised access to shared memory.

Blocking times are the result of tasks waiting to enter a monitor. Schedulability analysis requires that blocking times are statically predictable. This imposes a requirement on the way run-time environments implement monitors. For instance, when several tasks wait for the same monitor, it must be transparent in which order the run-time system grants access to the monitor.

(Req 12) Monitor implementations must be such that the maximum blocking time incurred by a task is bounded and can be calculated statically.

Unfortunately, task synchronisation does not interact well with priority-based scheduling. The problem is that blocking may invert task priorities. In order to prevent priority inversion, run-time environments must adopt a synchronisation protocol. Schedulability analysis can cope with the priority inheritance protocol and the priority ceiling protocol. The latter has the additional advantage that, on single core machines, it does not only minimise priority inversion, but also prevent deadlocks.

(Req 13) Run-time environments must support priority inheritance or the priority ceiling protocol, in order to bound the maximum duration of priority inversion due to synchronised access of shared resources.

The final quantity that contributes to response times is *interference by non-task entities and run-time mechanism*. This gives rise to the following requirement.

(Req 14) Run-time environments must ensure that the overhead of interrupts is bounded.

In general, interference by non-task entities is hard to predict statically. Furthermore, in disagreement with our technology-neutral approach, this quantity is often very platform-specific. For these reasons, we require that run-time environments support run-time monitoring of execution times, in order to detect execution time overruns and deadline misses at run-time. This makes it possible to control hard-to-predict interferences. If violations are detected, run-time environments should invoke programmable violation handlers.

(Req 15) Run-time environments must support execution time monitoring in order to detect execution time overruns and deadline misses.

(Req 16) Run-time environments must support the registration and invocation of programmable execution time overrun and deadline miss handlers.

Last but not least, it is crucial that run-time environments provide fine-grained support for specifying and measuring time (e.g., for specifying and enforcing periods, deadlines, and execution times).

(Req 17) Run-time environments must support a time-zone-independent, monotonically increasing, absolute clock. It is desirable that the granularity of time be as fine as possible.

Real-time programs require that tasks delay the release of their next job until a certain absolute time.

(Req 18) Run-time environments must support the specification and enforcement of absolute delays (i.e., delays that are specified by a point in time rather than a duration relative to the point of "invoking" the delay).

Schedulability analysis also makes the following assumption on dependencies between threads.

(Req 19) Generated programs must be such that each job has a single activation gate. The synchronisation and blockage of a thread that depends on other threads must be limited and statically computable.

2.2 Requirements for Predictable Memory Consumption and Memory Access

The goal of a memory analysis is to determine an upper bound on the memory demand for an application.

(Req 20) Run-time environments or generated programs should be such that it is possible to statically determine an upper bound on the amount of memory that is required to run the program.

In some cases, a finer analysis that determines memory demands for particular system components (instead of memory demands for the whole program) is also desirable. Such program components could, for instance, be threads or thread groups.

Unfortunately, it is hard to formulate specific, technology-neutral requirements that facilitate the prediction of memory consumption. We mention a number of technology-specific guidelines that help predicting memory usage:

- For predictability of heap size, programs should allocate all dynamic memory in an initialisation phase. Ravenscar Ada enforces such a memory management discipline.
- For predictability of stack size, programs should avoid recursion.
- For predictability of scoped and immortal memory size in Java/RTSJ, programs should avoid dynamic memory allocation inside unbounded loops[1] and unbounded recursion.
- For predictability of heap size in a Java VM with work-based real-time garbage collector, it is necessary to predict an upper bound on the amount of reachable memory. To this end, programs should avoid dynamic memory allocation in unbounded loops and unbounded recursion.

These guidelines are overly restrictive. For instance, the second guideline on avoiding recursion is not needed for stack usage predictability: it is also possible to predict stack usage for recursive programs, if one can prove a bound on recursion depths. Furthermore, these guidelines are not technology-neutral, because different programming environments provide different kinds of temporary memory (only stack for Ravenscar Ada; stack, scoped memory and heap for Java/RTSJ). The lack of static and technology-neutral memory analysis methods calls for run-time enforcement of memory budgets.

(Req 21) Run-time environments should monitor memory usage and enforce memory budgets (including heap, stack, pool and static memory, both code and data).

In order to avoid the need for predicting virtual memory consumption, it is necessary that run-time environments can configure and limit its use.

(Req 22) Run-time environments should offer the possibility to limit or avoid the use of virtual memory.

An important requirement that does not concern the consumption of memory, but rather its predictable use, is the need for atomic memory access. This requirement is a basis for predictability of program behaviour in general.

(Req 23) Run-time environments must support atomic memory access.

[1] An unbounded loop does not necessarily have to be non-terminating. It can also be unbounded, if its exit condition depends on user input.

2.3 Requirements for Dependability

This section presents the core set of requirements that should be enforced at run-time to support the types of dependability analysis techniques identified in D3.1. Each technique is actually based on a set of assumptions that allow/facilitate the modelling construction and solution process, and obviously the significance of the produced analysis results is strictly related to the truthfulness of such assumptions, i.e., to the extent to which they are verified during the real system's behaviour. Therefore, it is of utmost importance to identify appropriate run-time mechanisms that allow to enforce the identified requirements, thus improving the meaningfulness of the results produced by the static analysis methods. All the run-time mechanisms for dependability that will be described in Section 3 go in this direction.

Nevertheless it is worthwhile to clarify that no existing run-time mechanism (or combination of them) can assure that all the identified dependability requirements are fully met, mainly for the following two reasons:

- There is a large gap between the abstraction level of the static analysis methods and the concrete run-time environments;
- Most requirements for dependability are system level requirements, whereas the run-time environments supplied in CHESS are software level.

Despite such difficulties, it still is important to collect the dependability requirements so as to provide a direction to the partners that supply run-time environments. More concise and refined requirements will be developed throughout the remainder of the project, driven by the use cases.

Run-time requirements for state-based analysis In this paragraph we sketch a set of assumptions at the base of the state-based analysis models.

(Req 24) Run-time environments should provide support for *fault diagnosis* mechanisms that identify and record the cause(s) of error(s) in terms of both location and type, and possibly notify the presence of an unexpected fault (i.e., not considered at model-level). E.g.: alpha-count to discriminate between transient and permanent faults.

(Req 25) Run-time environments should provide support for *error detection* techniques that identify the presence of an error, and possibly notify the presence of an unexpected error (i.e., not considered at model-level). E.g.: parity check to detect 1 bit flip errors.

(Req 26) Run-time environments should provide support for *failure detection* techniques that identify the presence of a failure mode, and possibly notify the presence of an unexpected failure mode (i.e., not considered at model-level). E.g.: failure detectors to check halt failures.

(Req 27) Run-time environments should provide support to monitor the conformance of the *fault-error-failure chains* perceived during operation with respect to the fault-error-failure chains defined at model-level, possibly raising a notification.

(Req 28) Run-time environments should provide support for mechanisms improving the *isolation* between non interacting components (i.e., components not having functional relations). E.g.: adoption of MMU.

(Req 29) Run-time environments should provide support for implementing *restarting mechanisms* to repair/replace specific (hw/sw) components. E.g.: restart for sw rejuvenation.

(Req 30) Run-time environments should provide support for *reconfiguration actions* through mechanisms that allow either switches in spare components or reassign tasks among non failed components.

Requirements for failure propagation and transformation analysis A Failure Propagation and Transformation analyst is able to provide the failure behaviour description for each component. Each component can be analysed in isolation from the rest of the system.

(Req 31) Systems must be such that the software architecture is statically determined. Threads and communications channels must not be created and destroyed dynamically.

(Req 32) The failure propagation analyst should consider all possible failures on input. In case the analyst forgets to specify the behaviour of an input failure condition it is assumed that this failure is always propagated as is.

(Req 33) Failure propagation analysis assumes that failure behaviour on output is deterministic.

Run-time techniques for supporting dependability The role played by the run-time supports from the dependability perspective is twofold. On the one hand, it should allow to enforce the basic modelling assumptions, i.e., to "validate" the models. On the other hand, it should support mechanisms to facilitate the achievements of the required system dependability properties. This paragraph enumerates run-time techniques that support dependability. These are described in detail in D3.3.

Fault prevention

- Safe programming languages
- Code signing and access control
- Static code analysis
- Dynamic code analysis
- Tracing and checking
- Error injection
- Design under test configuration
- Case generation and automation
- Result analysis and calculation
- Load Balancing
- Flow control

Error detection

- Sanity checks
- Reversal checks
- Control and monitor
- Recovery blocks
- Watchdog timers
- Replication checks
- Control flow monitoring
- Cyclic Redundancy Codes
- Diagnostic checks for RAM, word-oriented memory, flash memory and CPU

Error handling

- Transaction-based roll-back recovery
- Hardware redundancy

- Threshold-based techniques for diagnosis
- Timing and space partitioning
- System reconfiguration

Fault removal

- Preventive maintenance
- Corrective maintenance

2.4 Further Run-time Features Desired by End Users

This section collects additional features for run-time monitoring and property enforcement that end users found desirable. Some of these features are specific to certain end user domains, so it is not critical that these features must be provided by all target run-time environments.

While static analysis is preferable if possible, run-time enforcement can be useful for several reasons:

- *Robustness against hard-to-predict conditions or events.*

  Regarding schedulability analysis, it was discussed in Section 2.1 that certain interferences from the environment are hard to predict statically. Run-time property enforcement increases robustness against hard-to-predict conditions and events.

- *Complementing static analysis.*

  Some interesting non-functional properties are not sufficiently covered by the static analysis methods explored by CHESS partners. An example is memory usage analysis. Run-time property enforcement (e.g., enforcement of memory budgets) helps to cover such properties to some extent.

- *Dealing with untrusted and legacy code.*

  Run-time property enforcement helps to widen the scope of CHESS to systems that make use of untrusted code that is not generated from CHESS models. Untrusted code could, for instance, be invoked by calling external libraries. When using untrusted components, programmers could write models of such components. Such models would reflect the programmers' assumptions of the component behaviour. While the conformance of untrusted code with such models could not be verified statically (e.g., because the source code is unavailable), it would be possible to ensure conformance through run-time monitoring.

Further features desired by all end user domains The CHESS partners find it generally useful to enforce *per-component resource budgets* at run-time. Resources can for instance be time or memory. Typically, one does not only want to assign a single resource budget to the entire application, but different resource budgets to different program components, possibly with the intent of preventing that some program components monopolise all resources. For instance, in systems where single program components "own" multiple threads, resource budgets per thread group are useful.

(Req 34) Run-time environments should support attributing resource budgets to groups of active entities (e.g., task groups or thread groups), and support monitoring/enforcing per-group budgets at run-time.

Further features desired by the telecommunications domain

(Req 35) *Throughput.* Measured by properties such as bits/second, message/second, package/second. The analysis should consider control plane, user plane and data plane end-to-end.

(Req 36) *Execution Capacity with respect to CPU load.* For the telecom domain this can be measured in terms of number of processes, call-setup/second, signals/second, connections/second.

(Req 37) *Latency.* Measured through properties such as call-setup time, message transfer, task scheduling and completion, interface specifications (services), payload. Latency is also considered as both latency to set up a connection and also latency for the data plane/payload parts.

(Req 38) *In Service Performance (ISP).* By enforcing system down time and Mean-Time-Between-Failures (MTBF) properties.

Further features desired by the space domain

(Req 39) Run-time environments should support the possibility to disable all run-time checks.

2.5 End User Evaluation of Requirements

In this section, end users evaluate the requirements that were imposed by the capabilities of static analysis methods and their underlying models. The purpose is to identify requirements that may be too restrictive for certain end user domains and find ways of resolving such issues.

Telecommunication Domain

Requirement	Discussion	Evaluation
(Req 1) **(Req 2)** **(Req 3)** *(periodic and sporadic tasks)*	The current dominant approach in development of systems in the telecommunication domain is that a priority-based approach is used; however, no timing specifications regarding period, execution time and deadline is defined per tasks (processes). These aspects are implicit in the design of the system in the sense that for example the appropriate execution time for a task is determined through different tests and using the experience and knowledge that engineers have about the system. If it is noticed that a task is taking too much time and cannot fulfil the system requirements, engineers re-implement it in a way to fit the system without explicitly defining its execution time or deadline. Harder and more rigorous bounds on execution times will, however, be beneficial in the telecommunications domain.	non-conflicting
(Req 4) **(Req 5)** *(no dynamic thread creation, no thread termination)*	Dynamic thread creation is used a lot in telecommunication systems. Besides creating threads dynamically, threads are also killed and terminated in the system. Hence, regarding the validity of telecommunication systems. Besides creating threads dynamically, thread are also killed and terminated in the system. Hence regarding the validity of analysis results, this aspect should also be taken into account besides dynamic thread creation. Also as an example, in terms of redundancy and fault tolerance mechanisms, it is not economical to have processes on backup nodes running when not needed. On the other hand, theoretically, if a system is given enough time to run, it would reach a stable state in the sense that all threads are created and memory from pool is allocated. In summary, prevention of dynamic thread creation and termination has conflicts with the requirements of telecommunication systems in general and possible relaxations are being discussed with analysis teams (e.g., a mode-based programming model). In the scope of CHESS, a subsystem in which use of static threads (initialised at the system start-up and not allowed to be terminated) is feasible can be considered. The issue of extending the approach to other parts of a telecommunication system should later be investigated more thoroughly.	too restrictive, but it may be feasible that a special subsystem satisfying this requirement be selected for CHESS prototype
(Req 8) *(time-bounded management tasks)*	One scenario could be that a flash memory has bad blocks so the time taken to read/write can be different depending on the number of bad blocks. However, the worst-case scenarios can be calculated based on the user manual of the flash memory for read and write accesses. Another important scenario in telecommunication domain is about data traffic and communication lines. There can be errors in transfer and collisions can occur which results in numerous re-sends of the original packets plus additional control packets.	non-conflicting

(Req 19) *(single activation gate)*	This requirement can be problematic in some scenarios and the context in which it is applied is very important. As an example, we considered a client-server scenario with multiple clients and a single server. This scenario seemed to violate the requirement for a single activation gate, because multiple clients can activate the server. However, after discussion with the analysis teams, we found that this scenario typically does not violate the requirement, because there can be a single activation gate on the server side, which can be activated by multiple clients. (This activation gate would consist of the server waiting for some message from some client.) However, we cannot assert and assume for sure that the requirement for a single activation gate can always be satisfied in telecommunication systems. An exhaustive and detailed investigation of the use case is required to determine this, which is best carried out once prototypes have been developed.	may be problematic in some scenarios, but can be verifiable in a subsystem e.g. chosen for CHESS prototype

All other requirements are **non-conflicting** with the needs of the telecommunication domain.

Space Domain

Requirement	Discussion	Evaluation
(Req 13) *(priority inheritance or priority ceiling protocol)*	For us PCP is mandatory, PI optional.	PCP is mandatory
(Req 19) *(single activation gate)*	On our architecture, this rule might entail the creation of numerous threads, which is not really compatible with our HW. At this point, it is hard to evaluate whether this is blocking or can be overcome. This issue will therefore be addressed once prototypes have been developed.	may be problematic

All other requirements are **non-conflicting** with the needs of the space domain.

Railway Domain

All requirements are **non-conflicting** with the needs of the railway domain.

2.6 Summary of Requirements

Requirements for Schedulability Analysis

ID	Requirement
(Req 1)	Run-time environments must provide means to implement periodic and sporadic tasks. Both program-driven and interrupt-driven sporadic tasks must be supported.
(Req 2)	Run-time environments must enforce periodic job releases for periodic tasks.
(Req 3)	Run-time environments must enforce MIATs between job releases for sporadic tasks.
(Req 4)	Run-time environments must provide means to implement systems where all threads (i.e., the run-time implementations of tasks) are created in a system initialisation phase, and no more threads are dynamically created thereafter.
(Req 5)	Run-time environments must provide means to implement systems where all threads do not freely terminate.
(Req 6)	Run-time environments must ensure that each job has an associated worst-case execution time.
(Req 7)	Run-time environments must ensure that, for each job, the resources that are needed for executing the instruction are predictable (e.g., CPU, network and communication resources, thread-shared memory).
(Req 8)	Run-time environments must ensure that all management tasks that operate underneath the application software (e.g., automatic garbage collection) are time-bounded.
(Req 9)	Run-time environments must support fixed-priority preemptive scheduling. The association between threads and priorities must be static (except for dynamic priority changes due to priority inheritance).
(Req 10)	Run-time environments may optionally support earliest-deadline-first scheduling.
(Req 11)	Run-time environments must support monitors as a mechanism for synchronised access to shared memory.
(Req 12)	Monitor implementations must be such that the maximum blocking time incurred by a task is bounded and can be calculated statically.
(Req 13)	Run-time environments must support priority inheritance or the priority ceiling protocol, in order to bound the maximum duration of priority inversion due to synchronised access of shared resources.
(Req 14)	Run-time environments must ensure that the overhead of interrupts is bounded.
(Req 15)	Run-time environments must support execution time monitoring in order to detect execution time overruns and deadline misses.
(Req 16)	Run-time environments must support the registration and invocation of programmable execution time overrun and deadline miss handlers.
(Req 17)	Run-time environments must support a time-zone-independent, monotonically increasing, absolute clock. It is desirable that the granularity of time be as fine as possible.
(Req 18)	Run-time environments must support the specification and enforcement of absolute delays (i.e., delays that are specified by a point in time rather than a duration relative to the point of "invoking" the delay).
(Req 19)	Generated programs must be such that each job has a single activation gate. The synchronisation and blockage of a thread that depends on other threads must be limited and statically computable.

The most important concerns from specific end user domains:

(Req 4) For the telecommunications domain, this requirement is too restrictive. A relaxed programming model that is more acceptable for telecommunications is a mode-based model that allows

thread creation in certain execution modes (e.g., reconfiguration phases) and disallows thread creation otherwise. Such a programming model poses a greater challenge on the analysis methods. This topic is currently under discussion.

(Req 19) For the space and telecommunication domains, this requirement may be problematic for some programs where the requirement may lead to a large number of threads. It is hard to evaluate at this point in how far this can be overcome. This issue will therefore be evaluated further once prototypes have been developed.

Requirements for Predicting Memory Consumption and Memory Access

ID	Requirement
(Req 20)	Run-time environments or generated programs should be such that it is possible to statically determine an upper bound on the amount of memory that is required to run the program.
(Req 21)	Run-time environments should monitor memory usage and enforce memory budgets (including heap, stack, pool and static memory, both code and data).
(Req 22)	Run-time environments should offer the possibility to limit or avoid the use of virtual memory.
(Req 23)	Run-time environments must support atomic memory access.

Requirements for Dependability

ID	Requirement
(Req 24)	Run-time environments should provide support for *fault diagnosis* mechanisms that identify and record the cause(s) of error(s) in terms of both location and type, and possibly notify the presence of an unexpected fault (i.e., not considered at model-level). E.g.: alpha-count to discriminate between transient and permanent faults.
(Req 25)	Run-time environments should provide support for *error detection* techniques that identify the presence of an error, and possibly notify the presence of an unexpected error (i.e., not considered at model-level). E.g.: parity check to detect 1 bit flip errors.
(Req 26)	Run-time environments should provide support for *failure detection* techniques that identify the presence of a failure mode, and possibly notify the presence of an unexpected failure mode (i.e., not considered at model-level). E.g.: failure detectors to check halt failures.
(Req 27)	Run-time environments should provide support to monitor the conformance of the *fault-error-failure chains* perceived during operation with respect to the fault-error-failure chains defined at model-level, possibly raising a notification.
(Req 28)	Run-time environments should provide support for mechanisms improving the *isolation* between non interacting components (i.e., components not having functional relations). E.g.: adoption of MMU.
(Req 29)	Run-time environments should provide support for implementing *restarting mechanisms* to repair/replace specific (hw/sw) components. E.g.: restart for sw rejuvenation.
(Req 30)	Run-time environments should provide support for *reconfiguration actions* through mechanisms that allow either switches in spare components or reassign tasks among non failed components.
(Req 31)	Systems must be such that the software architecture is statically determined. Threads and communications channels must not be created and destroyed dynamically.
(Req 32)	The failure propagation analyst should consider all possible failures on input. In case the analyst forgets to specify the behaviour of an input failure condition it is assumed that this failure is always propagated as is.

(Req 33)	Failure propagation analysis assumes that failure behaviour on output is deterministic.

Further Features Desired by End Users

ID	Requirement	Source
(Req 34)	Run-time environments should support attributing resource budgets to groups of active entities (e.g., task groups or thread groups), and support monitoring/enforcing per-group budgets at run-time.	all
(Req 35)	*Throughput.* Measured by properties such as bits/second, message/second, package/second. The analysis should consider control plane, user plane and data plane end-to-end.	telecom
(Req 36)	*Execution Capacity with respect to CPU load.* For the telecom domain this can be measured in terms of number of processes, call-setup/second, signals/second, connections/second.	telecom
(Req 37)	*Latency.* Measured through properties such as call-setup time, message transfer, task scheduling and completion, interface specifications (services), payload. Latency is also considered as both latency to set up a connection and also latency for the data plane/payload parts.	telecom
(Req 38)	*In Service Performance (ISP).* By enforcing system down time and Mean-Time-Between-Failures (MTBF) properties.	telecom
(Req 39)	Run-time environments should support the possibility to disable all run-time checks.	space

3 Feasibility of Requirements

Outline The target programming languages of the CHESS model transformation chains are Ada Real-time, Java/RTSJ, and C/C++. The C and C++ programs will make use of APIs provided by the real-time operating systems Real-time Linux and OSE. This section discusses how the requirements that were presented in Section 2 can be realised in each of these target environments. Each of the subsections begins with a short overview of the respective run-time environment. These overviews are followed by feasibility evaluations for the requirements.

Concerning feasibility of dependability requirements As discussed in Section 2.3, the requirements for dependability are less concise than the ones for predictability. More concise dependability requirements will evolve when the use cases are developed further. Consequently, the sections that address dependability (Sections 3.1.4, 3.2.4, 3.3.4, 3.4.4) describe run-time mechanisms for supporting dependability without evaluating the feasibility of concise dependability requirements one by one. Guided by the requirements from Section 2.3, these sections describe run-time mechanisms that are already provided by the supported run-time environments, or could be provided within CHESS towards enhancing dependability.

3.1 Ada for Real-Time Systems

Ada 83 supplied facilities for real-time software programming. The unit program "task" was the main entity as unit of concurrency with the rendezvous as communication mechanism.

Ada 95 enhanced the real-time features with more efficient mechanisms for synchronisation and communication as the "protected object" and with the introduction of several annexes for the development of real-time embedded systems, as Annex C for systems programming, Annex D for real-time features, Annex E for distributed Systems and Annex H for High Integrity Systems.

Ada 2005 enriches the Annex D with new dispatching policies (Earliest Deadline First (EDF), Round Robin (RR)) and with new libraries for execution time control, timing events.

Ada 2005 [1] now includes the Ravenscar profile for high-integrity systems which is a part of the Annex D. This profile implies a set of restrictions in order to meet all requirements to support the development of such systems as the determinism, the schedulability analysis, the memory-boundedness, the software certification. These restrictions are added to those required by the Annex H.

3.1.1 Overview of the Ada Real Time and the Ravenscar profile

What follows is a summary of real-time features and restrictions defined by the Ada Ravenscar profile which is the targeted Ada execution platform.

Memory management

- Dynamic allocation is not allowed;
- No implicit allocation;
- Storage for tasks is statically pre-allocated.

Clock and Time

- Relative delay is not allowed;
- The clause "D.8 Monotonic Time" defines the language-defined library Ada.Real_Time. This package defines a high-resolution and monotonic clock package.

Schedulers

- Explicit dynamic priority is not allowed;
- The scheduling is facilitated by the following mechanisms;
  - Preemptive dispatching;
  - Round Robin dispatching (time slicing);
  - First-In-First-Out within priority;
  - Earliest Deadline First (EDF).
- Priority Ceiling Locking specifies the interactions between priority task scheduling and protected object ceilings:
  - The ceiling priority of a protected object is an upper bound on the active priority a task can have.
- Task termination is not allowed.

Schedulable Objects

- Schedulable objects: The Ada unit program "task" is the schedulable object. Periodic, aperiodic and sporadic task activation patterns may be supported. Asynchronous event handlers can be bound to both program-triggered and environment-triggered events.
- Release parameters: (if required)
  - Deadline: the initial deadline to a task;
  - Cost overrun handler: use of library package Ada.Execution_Time.Timers;
  - Deadline miss handler: use of library package Ada.Real_Time.Timing_Events;
  - Start time for task: at the beginning of the main unit;
  - Period for periodic task: use of delay_until_statement (Interval between two releases);
  - Minimal Inter-Arrival Time for sporadic task: use of delay_until_statement (release time + MIAT) enforced by construction.
- Scheduling parameters;
  - Priority: the range is implementation-defined.
- Memory parameters;
  - Memory budget: storage size for amount of storage to be reserved for the execution of a task.
- Processing groups: library package Ada.Execution_Time.Group_Budgets allows several tasks to share a budget and provides means whereby action can be taken when the budget expires.

Task synchronisation

- The forms of synchronisation allowed by the Ravenscar Profile are as follows:
  - the activation of a task;

- a call on a protected subprogram of a protected object, providing exclusive read-write access, or concurrent read-only access to shared data;
- a call on an entry of a protected object, allowing for asynchronous communication with one tasks;
- synchronisation through suspension object.

Here are the restrictions required by the Ravenscar profile. Some of them do not allow some real time features listed before:

The run-time profile Ravenscar is equivalent to the following set of pragmas:

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
        No_Abort_Statements,
        No_Dynamic_Attachment,
        No_Dynamic_Priorities,
        No_Implicit_Heap_Allocations,
        No_Local_Protected_Objects,
        No_Local_Timing_Events,
        No_Protected_Type_Allocators,
        No_Relative_Delay,
        No_Requeue_Statements,
        No_Select_Statements,
        No_Specific_Termination_Handlers,
        No_Task_Allocators,
        No_Task_Hierarchy,
        No_Task_Termination,
        Simple_Barriers,
        Max_Entry_Queue_Length => 1,
        Max_Protected_Entries => 1,
        Max_Task_Entries => 0,
        No_Dependence => Ada.Asynchronous_Task_Control,
        No_Dependence => Ada.Calendar,
        No_Dependence => Ada.Execution_Time.Group_Budget,
        No_Dependence => Ada.Execution_Time.Timers,
        No_Dependence => Ada.Task_Attributes);
```

3.1.2 Feasibility of Requirements for Schedulability Analysis

This section describes if and how the requirements collected in Section 2.1 can be met. For each requirement, the feasibility is expressed in the following way:

- Feasibility based on the definition of Ada 2005.
- Equivalences or current limitations introduced by the current implementation of ObjectAda Raven implementation.

The Ravenscar model of tasking originated at the Eighth International Real-Time Ada Workshop, which met in Ravenscar, UK, in April 1997. A major result of this workshop was the “Ravenscar

profile", a restricted Ada 95 tasking model to meet the real-time requirements for schedulability analysis.

The 2005 revision of the Ada language [1] includes the specification of the Ravenscar profile.

The current ObjectAda Raven implementation by Atego (formely Aonix) [2] provides a subset of Ada 95 that complies with the Ravenscar profile.

Feasibility of (Req 1)

> Run-time environments must provide means to implement periodic and sporadic tasks. Both program-driven and interrupt-driven sporadic tasks must be supported.

Periodic task representation. A cyclic task pattern may be represented by a discriminated task type with a *know_discriminant_part* which requires the period of the task for any task objects declaration. The task body has an outermost infinite loop containing one *delay_until* statement.

```
task type Periodic_Task (Period : ...) is
end Periodic_Task;

task body Periodic_Task is
   Next_Release : Time := Clock;
begin
   loop
      delay until Next_Release;
      ... -- perform some actions

      Next_Release := Next_Release + Period;
   end loop;
end Periodic_Task;
```

Sporadic task representation: program-driven (semaphore use case). A basic program-driven sporadic task pattern may be represented by a task type using a semaphore. The task body has an outermost infinite loop containing a call to the procedure

> Ada.Synchronous_Task_Control.Suspend_Until_True

on an object of the type

> Ada.Synchronous_Task_Control.Suspension_Object.

Such a pattern may be used when only a synchronisation point is needed, but no data passing.

```
Sporadic_Event : Ada.Synchronous_Task_Control.Suspension_Object;
task type Sporadic_Task is

end Sporadic_Task;
```

```
task body Sporadic_Task is

begin
  loop
    Ada.Synchronous_Task_Control.Suspend_Until_True(Sporadic_Event);
      ... -- perform some sporadic actions
  end loop;
end Sporadic_Task;
```

The language-defined private semaphore (suspension object) used above:

```
package Ada.Synchronous_Task_Control is
   pragma Preelaborate(Synchronous_Task_Control);
   type Suspension_Object is limited private;
   procedure Set_True(S : in out Suspension_Object);
   procedure Set_False(S : in out Suspension_Object);
   function Current_State(S : Suspension_Object) return Boolean;
   procedure Suspend_Until_True(S : in out Suspension_Object);
private
... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

Sporadic task representation: program-driven (protected object use case). An enhanced *program-driven* sporadic task pattern may be represented by a task type using a protected object. The task body has an outermost infinite loop containing a call to an entry of a protected object. This entry has a guard which is a private data of type Boolean. A program call to the protected procedure sets the guard to True leading to the release of the sporadic task. In such a pattern, data passing is possible.

```
protected type Event is
   procedure Signal;     -- May be Signal(Some_Data: in  Data)
   entry Wait;           -- May be Wait(Some_Data: out Data)
private
   Signal_Occured : Boolean := False;
end Event;

protected body Event is
   procedure Signal is
   begin
      ...
      Signal_Occured:= True;
   end Signal;
   entry Wait when Signal_Occured is
   begin
      ...
      Signal_Occured:= False;
   end Wait;
end Event;

Sporadic_Event : Event;
```

```
task type Sporadic_Task is
end Sporadic_Task;

task body Sporadic_Task is
   Next_Release : Ada.Real_Time.Time := Ada.Real_Time.Clock;
Begin
   ...
   loop
      Sporadic_Event.Wait;     -- may be Sporadic_Event.Wait(Some_Data)
      ... -- perform some sporadic actions

   end loop;
end Sporadic_Task;
```

Sporadic task representation: interrupt-driven. An *interrupt-driven* sporadic task is similar to the enhanced program-driven sporadic task, except that the protected procedure of the protected object is attached to an interrupt signal.

```
protected Interrupt_Event is
   entry Wait;
   procedure Signal;
private
   pragma Attach_Handler (Signal, <interrupt_id> );
   Current : Data;     -- Event data declaration
   Signal_Occured : Boolean := False;

end Interrupt_Event;

protected body Event is
   procedure Signal is
   begin
      ...
      Signal_Occured:= True;
   end Signal;
   entry Wait when Signal_Occured is
   begin
      ...
      Signal_Occured:= False;
   end Wait;
end Event;
```

Feasibility of (Req 2)

> Run-time environments must enforce periodic job releases for periodic tasks.

This is supported by the programming pattern presented above, using the delay-until operator. The semantics of the time type and the clock are as follows:

Ada RM D.8 Monotonic Time.

> Supports a high-resolution, monotonic clock package.

ObjectAda Raven Implementation.

> The Raven model supports only one Time type for use as the argument, namely, Ada.Real_Time.Time.

Feasibility of (Req 3)

> Run-time environments must enforce MIATs between job releases for sporadic tasks.

A sporadic task pattern with the Minimal Inter-Arrival Time (MIAT) enforcement may be represented by a task type as it has been previously described, but in this case, with an outermost infinite loop containing one *delay_until* statement to prevent any release *before* a given MIAT.

```
protected type Event is
   use Ada.Real_Time;
   procedure Signal;
   entry Wait (Release_Time : out Time);
private
   Signal_Occured : Boolean := False;
end Event;

protected body Event is
   procedure Signal is
   begin
      Signal_Occured:= True;
   end Signal;
   entry Wait (Release_Time : out Time) when Signal_Occured is
   begin
      Release_Time := Clock;
      Signal_Occured:= False;
   end Wait;
end Event;


Sporadic_Event : Event;
task type Sporadic_Task (MIAT : Ada.Real_Time.Time_Span) is
```

```
end Sporadic_Task;

task body Sporadic_Task is
   Next_Wait : Ada.Real_Time.Time := Ada.Real_Time.Clock;
Begin
   ...
   Loop
      delay until Next_Wait;

      ... -- wait statement for a sporadic event only after the release

      ... -- perform some sporadic actions

      Next_Wait := Release_Time + MIAT;
   end loop;
end Sporadic_Task;
```

Feasibility of (Req 4)

> Run-time environments must provide means to implement systems where all threads (i.e., the run-time implementations of tasks) are created in a system initialisation phase, and no more threads are dynamically created thereafter.

Ada 05 RM, H.6 Pragma Partition_Elaboration_Policy. This clause defines a pragma for user control over elaboration policy. Syntax:

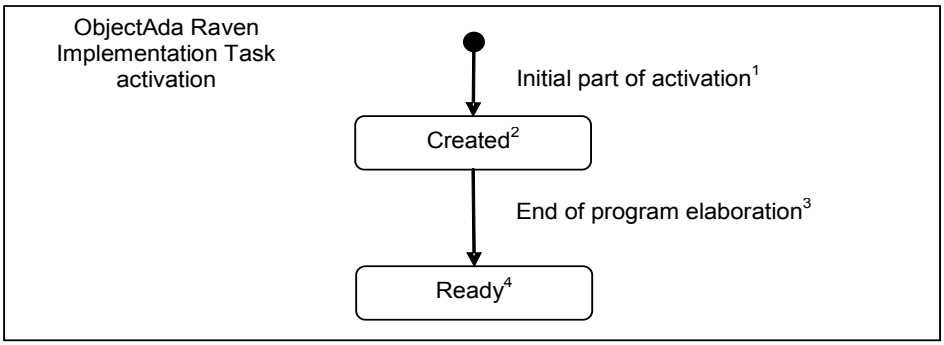
pragma Partition_Elaboration_Policy (policy_identifier);

[...]

If the partition elaboration policy is sequential, then task activation and interrupt attachment are performed in the following sequence of steps:

- The activation of all library-level tasks and the attachment of interrupt handlers are deferred until all library units are elaborated.
- The interrupt handlers are attached by the environment task.
- The environment task is suspended while the library-level tasks are activated.
- The environment task executes the main subprogram (if any) concurrently with these executing tasks.

ObjectAda Raven Implementation. The ObjectAda Raven tasking implementation leads to a similar behaviour, as shown in the following diagram:



1. *elaboration of the declarative part of the task body*
2. *the task is consistent and fully elaborated*
3. *after execution of the elaboration code of all library units, but before calling the main subprogram*
4. *state of a task when it is no longer suspended. The task, however, will not execute whilst all the available processor resource can be used by higher priority ready tasks.*

Feasibility of (Req 5)

> Run-time environments must provide means to implement systems where all threads do not freely terminate.

A non-terminating task pattern may be represented by a task type with a task body containing an outermost infinite loop containing itself at least one suspension statement.

```
task type Non_Terminating_Task (...) is

end Non_Terminating _Task;

task body Non_Terminating_Task is
   ...
begin
   loop
      ... - a suspension statement
      ... -- perform some actions

   end loop ;
end Non_Terminating_Task ;
```

Ada 05 RM, D.7 Tasking Restrictions. The restriction No_Task_Termination is used with the pragma Restrictions. All tasks are non-terminating. It is implementation-defined what happens if a task attempts to terminate. If there is a fall-back handler (see C.7.3) set for the partition it should be called when the first task attempts to terminate.

The restriction No_Abort_Statements is applied with the pragma Restrictions.

ObjectAda Raven Implementation. Task termination is a bounded error condition in the Ravenscar profile. The restriction No_Task_Termination is detected at run time as bounded error. In the event of task termination, including the environment task, the effect of the bounded error is to call the user-defined procedure System.Raven_Instrumentation.Task_Controller.Task_Termination_Handler followed by permanent suspension of the task if the called user-defined procedure returns. Other tasks in the program are not affected.

The restriction, No_Abort_Statements, is detected at compile time [RM D.7]. There are no abort_statements, and there are no calls to Task_identification.Abort_Task. The restriction No_Abort_Statements ensures that tasks cannot be aborted.

Feasibility of (Req 6)

> Run-time environments must ensure that each job has an associated worst-case execution time.

Ada RM, Annex D (normative). Real-Time Systems: This Annex specifies additional characteristics of Ada implementations intended for real-time systems software. To conform to this Annex, an implementation shall also conform to the Systems Programming Annex.

Metrics: The metrics are documentation requirements; an implementation shall document the values of the language-defined metrics for at least one configuration of hardware or an underlying system supported by the implementation, and shall document the details of that configuration.

ObjectAda Raven Implementation. Annex D, Real-Time systems: Metrics of the implementation are discussed in the Reference manual of the ObjectAda Raven implementation. The metrics have been measured on a specific board.

Example: The execution of a task can be preempted by the implementation's processing of delay expiration. The value is 20 microseconds for the eVAB-695E board.

Feasibility of (Req 7)

> Run-time environments must ensure that, for each job, the resources that are needed for executing the instruction are predictable (e.g., CPU, network and communication resources, thread-shared memory).

See Feasibility of **(Req 6)** for CPU.

Feasibility of (Req 8)

> Run-time environments must ensure that all management tasks that operate underneath the application software (e.g., automatic garbage collection) are time-bounded.

Ada RM D.7, Tasking Restrictions. No_Implicit_Heap_Allocations: There are no operations that implicitly require heap storage allocation to be performed by the implementation. The operations that implicitly require heap storage allocation are implementation defined.

ObjectAda Raven Implementation. Static memory model: These restrictions ensure that the total memory requirements beyond that allocated for global objects of the application are statically known and, in the case of standard storage pools, cannot become exhausted due to fragmentation or leakage, leading to Storage_Error exception.

Static_Storage_Size: The expression for pragma Storage_Size is static. The restriction Static_Storage_Size requires any value used in pragma Storage_Size to be static. This value controls the amount of storage reserved for each task stack. Note that the user must reserve storage in the Board Support Package for allocation of all of the task stacks.

No Standard_Storage_Pools: There are no uses of the standard storage pools. Each allocator references a user-defined pool. The soft restriction No_Standard_Storage_Pools prohibits the declaration of pool-specific access types that use a standard storage pool for the storage of the allocated objects. If the application requires use of Ada allocators and Ada.Unchecked_Deallocation to create and destroy objects dynamically, this can be achieved by writing an application-specific heap manager as an extension of System.Root_Storage_Pool [RM 13.11]. Note that predefined operations which make implicit use of a standard storage pool are not supported by the Raven subset.

Annex M, storage management: There are no cases in which the implementation dynamically allocates heap storage for a purpose other than the evaluation of an allocator.

Feasibility of (Req 9)

> Run-time environments must support fixed-priority preemptive scheduling. The association between threads and priorities must be static (except for dynamic priority changes due to priority inheritance).

Static priority model (Ada RM D1-7). A Priority pragma is allowed only immediately within a task_definition, a protected_definition, or the declarative_part of a subprogram_body (i.e the main subprogram).

```
task type Periodic_Task (Priority : System.Priority) is
   pragma Priority (Priority);
end Periodic_Task;
```

```
protected type Event(Ceiling : System.Priority) is
   pragma Priority(Ceiling);
   procedure Signal;
   entry Wait (Release_Time : out Time);
private
   Occurred : Boolean := False;
end Event;
```

Ada RM, D.1 Task Priorities. This clause specifies the priority model for real-time systems. In addition, the methods for specifying priorities are defined.

Syntax: The form of a pragma Priority is as follows: pragma Priority(expression); The form of a pragma Interrupt_Priority is as follows: pragma Interrupt_Priority[(expression)];

Ada 05 RM, D.7 Tasking Restrictions. The restriction No_Dynamic_Priorities is used with the pragma Restrictions.

ObjectAda Raven Implementation. No_Dynamic_Priorities, [RM D.7]: There are no semantic dependencies on the package Dynamic_Priorities. The restriction No_Dynamic_Priorities prevents use of the predefined package Ada.Dynamic_Priorities, thereby ensuring that the priority assigned at task creation is unchanged during the task's execution, except when the task is executing a protected operation, during which time it inherits ceiling priority.

Static_Priorities: All expressions specified as the parameter for pragmas Priority or Interrupt_Priority shall be static. The restriction Static_Priorities requires the assigned priority values within pragmas Priority and Interrupt_Priority to be static values (as in Ada83).

Feasibility of (Req 10)

> Run-time environments may optionally support earliest-deadline-first scheduling.

RM Ada 2005 D.2.2 Task Dispatching Pragmas. This clause allows a single task dispatching policy to be defined for all priorities, or the range of priorities to be split into subranges that are assigned individual dispatching policies. Syntax :

- The form of a pragma Task_Dispatching_Policy is as follows: pragma Task_Dispatching_Policy(policy_identifier);
- The form of a pragma Priority_Specific_Dispatching is as follows: pragma Priority_Specific_Dispatching (policy_identifier, first_priority_expression, last_priority_expression);

RM Ada 2005 D.2.6 Earliest Deadline First Dispatching. The deadline of a task is an indication of the urgency of the task; it represents a point on an ideal physical time line. The deadline might affect how resources are allocated to the task. This clause defines a package for representing the deadline of a task and a dispatching policy that defines Earliest Deadline First (EDF) dispatching. A pragma is defined to assign an initial deadline to a task.

ObjectAda Raven Implementation. Not implemented in the current release.

Feasibility of (Req 11)

> Run-time environments must support monitors as a mechanism for synchronised access to shared memory.

A protected object pattern is used to ensure mutually exclusive access to a shared resource, such as global data. Such protected object typically contains only protected subprograms as operations. Protected operations can only be executed by one task at a time (no multiple readers or writers). The following code listing shows an example use of a protected object.

```
protected Shared_Data is

    procedure Put (D : in Data);

    function Get return Data;

private
   Current : Data; Shared data declaration
end Shared_Data;

protected body Shared Data is

   procedure Put (D : in Data)
   begin
      Current := D;
   end Put;

   function Get return Data is
   begin
      return Current;
   end Get;

end Shared_Data;
```

Feasibility of (Req 12)

> Monitor implementations must be such that the maximum blocking time incurred by a task is bounded and can be calculated statically.

The Ravenscar restrictions require that, protected operations can only be executed by one task at a time (no multiple readers or writers), the body of the protected functions cannot call a blocking operation, and according to the ceiling protocol, the blocking time for the reader or the writer is predictable. (See also discussion of **(Req 11)** above.)

Feasibility of (Req 13)

> Run-time environments must support priority inheritance or the priority ceiling protocol, in order to bound the maximum duration of priority inversion due to synchronised access of shared resources.

Ada RM D.3 Priority Ceiling Locking. This clause specifies the interactions between priority task scheduling and protected object ceilings. This interaction is based on the concept of the ceiling priority of a protected object.

Syntax: The form of a pragma Locking_Policy is as follows.

pragma Locking_Policy(policy_identifier);

[. . .]

There is one predefined locking policy, Ceiling_Locking; this policy is defined as follows:

- Every protected object has a ceiling priority, which is determined by either a Priority or Interrupt_Priority pragma as defined in D.1, or by assignment to the Priority attribute as described in D.5.2. The ceiling priority of a protected object (or ceiling, for short) is an upper bound on the active priority a task can have when it calls protected operations of that protected object.

[. . .]

While a task executes a protected action, it inherits the ceiling priority of the corresponding protected object.

ObjectAda Raven Implementation. Deadlocks can be prevented by assigning a priority to each protected object that is just greater than the highest priority of all its calling processes, and the use of priority ceiling emulation within the run time via the locking policy Ceiling_Locking [RM D.3].

Feasibility of (Req 14)

> Run-time environments must ensure that the overhead of interrupts is bounded.

ObjectAda Raven Implementation. Example: The value is 34 microsecond for the eVAB-695E board.

Feasibility of (Req 15) and (Req 16)

> Run-time environments must support execution time monitoring in order to detect execution time overruns and deadline misses.
>
> Run-time environments must support the registration and invocation of programmable execution time overrun and deadline miss handlers.

The following pattern for deadline detection makes use of timing events and timing event handlers:

```
Deadline_Missed : Ada.Real_Time.Timing_Events.Timing_Event ; -- Ada05

task type Periodic_Task (Priority     : System.Priority;
                         Period       : Ada.Real_Time.Time_Span;
                         Max_Duration : Ada.Real_Time.Time_Span) is
   pragma Priority (Priority);
end Periodic_Task;

task body Periodic_Task is
   Next_Release : Ada.Real_Time.Time := Ada.Real_Time.Clock;
```

```
begin
   loop
      Ada.Real_Time.Timing_Events.Set_Handler
        (Deadline_Missed,
         Deadline,              -- Of type Ada.Real_Time.Time
         Deadline_Missed_Handler'Access);

      delay until Next_Release;
      ... -- perform some actions

      Next_Release := Next_Release + Period;
      Deadline := Next_Release + Max_Duration;
   end loop ;
end Periodic_Task;
```

Ada 2005 RM D.15 Timing Events This clause describes a language-defined package to allow user-defined protected procedures to be executed at a specified time without the need for a task or a delay statement. The type Timing_Event represents a time in the future when an event is to occur. The type Timing_Event_Handler identifies a protected procedure to be executed by the implementation when the timing event occurs. Such a protected procedure is called a handler.

The following code listing shows a pattern for detecting overruns of execution time budgets. This pattern makes use of execution time timers.

```
task type Periodic_Task (Priority : System.Priority;
                         Period   : Ada.Real_Time.Time_Span;
                         WCET     : Ada.Real_Time.Time_Span) is
   pragma Priority (Priority);
end Periodic_Task;

task body Periodic_Task is

   use Ada.Task_Identification;
   Task_Id : aliased constant Task_Id := Current_Task;
   WCET_Timer : Ada.Execution_Time.Timers.Timer (Task_Id'access) ;
   Timer_Cancelled : Boolean := False;

   Next_Release : Ada.Real_Time.Time := Ada.Real_Time.Clock;
begin
   loop
      delay until Next Time;

      Ada.Execution_Time.Timers.Set_Handler
        (WCET_Timer,
         WCET,
         WCET_Handler'Access);

      ... -- perform some actions
```

```
      Ada.Execution_Time.Timers.Cancel_Handler  -- Not needed D14.1 16/2
        (WCET_Timer,
         Timer_Cancelled);
      ...
      Next_Release := Next_Release + Period;
   end loop ;
end Periodic_Task ;
```

Ada 2005 RM D.14.1 Execution Time Timers. This clause describes a language-defined package that provides a facility for calling a handler when a task has used a defined amount of CPU time. The execution time of a task, or CPU time as it is commonly called, is the time spent by the system executing the task and services on its behalf.

Ravenscar restriction. The Ravenscar profile does not allow the use of this package.

pragma Restrictions (. . . No_Dependence => Ada.Execution_Time.Timers. . .)

ObjectAda Raven Implementation: The two following packages are not defined in Ada 95, therefore not supported by ObjectAda Raven:

- Ada.Real_Time.Timing_Events
- Ada.Execution_Time

Ada 2005's execution time timers provide the functionality requested by **(Req 15)**. Unfortunately, execution time timers are neither part of the definition of Ravenscar Ada nor of ObjectAda Raven. The functionality of an execution time timer is needed in CHESS, but it is yet to be determined how this will best be provided for the Ada technology in CHESS.

Feasibility of (Req 17) and (Req 18)

> Run-time environments must support a time-zone-independent, monotonically increasing, absolute clock. It is desirable that the granularity of time be as fine as possible.
>
> Run-time environments must support the specification and enforcement of absolute delays (i.e., delays that are specified by a point in time rather than a duration relative to the point of "invoking" the delay).

Ada RM, D.8 Monotonic Time. The real-time package is defined as follows:

```
package Ada.Real_Time is

   type Time is private;
   Time_First : constant Time;
   Time_Last : constant Time;
   Time_Unit : constant := implementation-defined-real-number;
```

```
   type Time_Span is private;
   Time_Span_First : constant Time_Span;
   Time_Span_Last : constant Time_Span;
   Time_Span_Zero : constant Time_Span;
   Time_Span_Unit : constant Time_Span;

   Tick : constant Time_Span;

   function Clock return Time;
   function "+" (Left : Time; Right : Time_Span) return Time;
   function "+" (Left : Time_Span; Right : Time) return Time;
   function "-" (Left : Time; Right : Time_Span) return Time;
   ...
   function Milliseconds (MS : Integer) return Time_Span;
   type Seconds_Count is range implementation-defined;
   procedure Split(T :in Time; SC : out Seconds_Count; TS : out Time_Span);
   function Time_Of(SC : Seconds_Count; TS : Time_Span) return Time;
private
... -- not specified by the language
end Ada.Real_Time;
```

Real time is defined to be the physical time as observed in the external environment. The type Time is a time type as defined by 9.6; values of this type may be used in a delay_until_statement. Values of this type represent segments of an ideal time line. The set of values of the type Time corresponds one-to-one with an implementation-defined range of mathematical integers.

[...]

The range of Time values shall be sufficient to uniquely represent the range of real times from program start-up to 50 years later. Tick shall be no greater than 1 millisecond. Time_Unit shall be less than or equal to 20 microseconds.

Ada 2005 RM, D.7 Tasking Restrictions. The restriction No_Relative_Delay is used with the pragma Restrictions. The restriction No_Dependence => Ada.Calendar is used with the pragma Restrictions.

No_Calendar: The restriction implies that there are no semantic dependencies on the package Ada.Calendar.

No_Relative_Delay: There are no Delay_Relative statements. The restriction No_Calendar, coupled with No_Relative_Delay, is introduced so that all timing is instead performed using the high precision afforded by the time type in package Ada.Real_Time. This time type has a precision of the same order of magnitude as the real-time clock device on the underlying board. In contrast, the time type in package Calendar, and predefined type Duration, generally have much coarser precision than the real-time clock which leads to inaccuracy in task release times.

ObjectAda Raven Implementation. The Raven model supports only one Time type for use as the argument in delay statement. Clocks and Timers: Package System.Board_Support.Clock provides the definition of the board's clock tick frequency for the run-time system. This code must match

the setting of the scalars for the real-time clock register and the general purpose timer register in timereVAB6 95E.s or timerdem32.s. By default, a 1 microsecond tick is generated for the eVAB-695E and DEM32. It is important to note that the Raven run time relies on the same tick frequency for both the real time clock and the general purpose timer. This package is used by the Raven run time to support the delay facility.

Feasibility of (Req 19)

> Generated programs must be such that each job has a single activation gate. The synchronisation and blockage of a thread that depends on other threads must be limited and statically computable.

Ada 2005 RM, D.7 Tasking Restrictions. The restriction No_Select_Statements is used with the pragma Restrictions. The restriction Max_Entry_Queue_Length = 1 is used with the pragma Restrictions. The restriction Max_Protected_Entries = 1 is used with the pragma Restrictions. The restriction Max_Task_Entries = 0 is used with the pragma Restrictions.

ObjectAda Raven Implementation. No Select Statements: There are no select statements.

Max_Entry_Queue_Depth = 1: Specifies the maximum number of tasks which may be queued concurrently per protected entry. For Raven, the value of Max_Entry_Queue_Depth is 1. The restrictions Max Protected Entries= 1 and Max_Entry_Queue_Depth = 1 ensure that at most one task can be suspended waiting on a closed entry barrier for each protected object which is used as a task synchronisation primitive. This avoids the possibility of queues of tasks forming, with the associated non- determinism of waiting time in the queue, and also avoids the possibility of there being more than one waiting task whose barrier becomes open simultaneously as the result of a protected action, with the associated non-determinism of selecting which protected action is executed next. Note that Max_Entry_Queue_Depth is checked at run time and so violation is a bounded error. When the restriction Max_Entry_Queue_Depth = 1 is in force, pragma QueueingPolicy ([RM DA]) has no effect.

Max Protected Entries = 1: [RM D.7] Specifies the maximum number of entries per protected type. The bounds of every entry family of a protected unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static. For Raven the value of Max_Protected_Entries is always 1.

Max Task Entries = 0: [RM D. 7J Specifies the maximum number of entries per task. The restrictions Max_Task_Entries=O and No_Select_Statements prohibit the use of Ada rendezvous for task synchronisation and communication. This ensures that these operations are achieved using only the two supported task synchronisation primitives: protected object entries and suspension objects, which both exhibit time-deterministic execution properties needed for static timing analysis.

3.1.3 Feasibility of Requirements for Predictable Memory Consumption and Access

Feasibility of (Req 20), (Req 21)

> Run-time environments or generated programs should be such that it is possible to statically determine an upper bound on the amount of memory that is required to run the program.
>
> Run-time environments should monitor memory usage and enforce memory budgets (including heap, stack, pool and static memory, both code and data).

Ada RM, 13.3 (61). A pragma Storage_Size specifies the amount of storage to be reserved for the execution of a task.

Syntax: The form of a pragma Storage_Size is as follows.

pragma Storage_Size(expression);

A pragma Storage_Size is allowed only immediately within a task definition. The expression of a pragma Storage_Size is expected to be of any integer type.

Ada RM, D.7 (8) Tasking Restrictions. No_Implicit_Heap_Allocations: There are no operations that implicitly require heap storage allocation to be performed by the implementation. The operations that implicitly require heap storage allocation are implementation defined.

Ada 2005 RM, D13.1. The run-time profile Ravenscar is equivalent to the following set of pragmas:

pragma Restrictions (. . . No_Implicit_Heap_Allocations, . . .);

ObjectAda Raven Implementation. Memory model: In order to meet the deterministic memory-size requirement necessary to ensure that the application cannot fail due to running out of dynamically-acquired memory, the Raven model requires all task stacks to have a static size (which can, of course, vary from task to task), and does not provide a default heap management system for dynamic acquisition of memory via allocators. However in recognition that some real-time systems do require the controlled allocation and deallocation of dynamic objects, user-defined heap management systems can be written using the Storage_Pools concept [RM 13.11], and one example of such storage pool that exhibits time-bounded allocation and deallocation, and cannot suffer from fragmentation, is provided in the Raven run time.

Restrictions:

Static memory model: These restrictions ensure that the total memory requirements beyond that allocated for global objects of the application are statically known and, in the case of standard storage pools, cannot become exhausted due to fragmentation or leakage, leading to Storage_Error exception.

Static_Storage_Size: The expression for pragma Storage_Size is static. The restriction Static_Storage_Size requires any value used in pragma Storage_Size to be static. This value controls the amount of storage reserved for each task stack. Note that the user must reserve storage

in the Board Support Package for allocation of all of the task stacks. Static_Storage_Size is a soft restriction which can be removed if a non-static expression is required in pragma Storage_Size.

No Standard_Storage_Pools: There are no uses of the standard storage pools. Each allocator references a user-defined pool. The soft restriction No_Standard_Storage_Pools prohibits the declaration of pool-specific access types that use a standard storage pool for the storage of the allocated objects. If the application requires use of Ada allocators and Ada.Unchecked_Deallocation to create and destroy objects dynamically, this can be achieved by writing an application -specific heap manager as an extension of System.Root_Storage_Pool [RM 13.11]. Note that predefined operations which make implicit use of a standard storage pool are not supported by the Raven subset (see the Ada 95 Reference Manual Notes for a full definition of unsupported features).

Feasibility of (Req 22)

> Run-time environments should offer the possibility to limit or avoid the use of virtual memory.

ObjectAda Raven Implementation: N/A

Feasibility of (Req 23)

> Run-time environments must support atomic memory access.

Note: The atomic memory access may be implemented as an object of a protected type.

Ada allows the use of atomic object if the hardware architecture allows such implementation for the object type.

Ada RM, C6 Shared Variable Control. This clause specifies representation pragmas that control the use of shared variables.

```
pragma Atomic(local_name);
pragma Atomic_Components(array_local_name);
```

For an atomic object (including an atomic component) all reads and updates of the object as a whole are indivisible. Atomic objects are objects for which the underlying hardware memory access allows indivisible load and store operations, such as integers.

3.1.4 Run-Time Mechanisms for Dependability

The Ada language provides dependability support for software in terms of reliability, availability, maintainability and safety (known as RAMS) but does not provide security support in terms of confidentiality, integrity and availability, from a point of view of unauthorised accesses. In terms of dependability means, the Ada language provides support for fault prevention (i.e., strong typing, information hiding, *pragma restrictions*) and fault tolerance (i.e., *Exception mechanism*). Some mechanisms may achieve at the same time fault prevention and fault tolerance (i.e., *sharing protocols*).

Such dependability means are mainly based on the identification of the errors at different stages of the software development.

For that, Ada 2005 Reference Manual gives the following *Classification of Errors*:

- *"Errors that are required to be detected prior to run time by every Ada implementation"*: i.e., errors detected at compile time when the context allows detecting any violation of the language legality rules. In such case, the program is not legal and such program cannot produce a binary executable, e.g. an illegal value assignment when the type of the value mismatches with the type of the targeted object.
- *"Errors that are required to be detected at run time by the execution of an Ada program"*: i.e., errors associated to the predefined exceptions. Those errors may be detected at compile time but does not prevent the production of the binary executable; implementation compiler may only report that the corresponding exception shall be raised at run time, e.g. an assignment of a value outside of the authorised range to an integer data raises an exception name Constraint_Error.
- *"Bounded errors"*: i.e., errors possibly not detected either at compile time or at run time but with a bounded range of possible effects. Leading necessarily if detected to a specific exception named Program_Error (Constraint_Error in some cases), e.g. a invocation to a potential blocking operation inside a protected action is a bounded error raising Program_Error if detected or leading to a deadlock if not.
- *"Erroneous execution"*: i.e., errors with not predictable effect, e.g. *"Program execution is erroneous if pragma Restrictions(No_Exceptions) has been specified and the conditions arise under which a generated language-defined run-time check would fail"* (Ada RM H.4 (26)).

Ada's exception mechanism An exception represents an kind of exceptional situation (Ada RM 11), i.e., when an error is detected at run-time; This error may be related to *predefined exceptions*, *standard library unit exceptions* or *user defined exceptions*. An occurrence of an exception leads to abandon the normal program execution, in response: when it occurs in a statement part, an exception handler may perform some actions otherwise the exception is propagated to the caller, as when this exception occurs in a declarative part.

Predefined Exceptions: The Ada language defines a set of predefined exceptions. These exceptions are raised when *language-defined checks* (see below) detect error situations. The Ada reference manual does not give the exhaustive list of these checks. However, the given list is to allow the suppression or the required effect of specific checks (by *pragma*). These checks are only a subset of all the language defined checks. The Annotated Ada Reference Manual says more about other checks, e.g. Ceiling_Check (if pragma Locking_Policy (Ceiling locking) is in effect).

Here is the list of predefined exceptions:

- Constraint_Error, .e.g. Division_Check for division by zero;
- Program_Error, e.g. Elaboration_Check for program body unit elaboration;
- Storage_Error, e.g. Storage_Check for space available for a task;
- Tasking_Error, e.g. run-time checks, not named by the standard;

Standard library unit exceptions: These exceptions are propagated out of a standard library unit when an exceptional situation occurs inside corresponding body unit; e.g. Data_Error is propagated when the element read cannot be interpreted as a value of the targeted data.

User defined exceptions: These exceptions describe exceptional situation related to the application logic; hence they are defined and raised by the user.

Language-defined checks:

> A language-defined check (or simply, a "check") is one of the situations defined by this International Standard that requires a check to be made at run time to determine whether some condition is true. A check fails when the condition being checked is false, causing an exception to be raised (Ada RM 11.5 (2)).

Here are the language-defined checks named by the standard:

- *Range_Check* leading to a Constraint_Error;
- *Overflow_Check* leading to a Constraint_Error;
- *Elaboration_Check* leading to a Program_Error;
- *Index_Check* leading to a Constraint_Error;
- *Division_Check* leading to a Constraint_Error;
- *Tag_Check* leading to a Constraint_Error;
- *Access_Check* leading to a Constraint_Error;
- *Discriminant_Check* leading to a Constraint_Error;
- *Length_Check* leading to a Constraint_Error;
- *Partition_Check* leading to a Constraint_Error;
- *Accessibility_Check* leading to a Program_Error;
- *Allocation_Check* leading to a Program_Error;
- *Reserved_Check* leading to a Program_Error;
- *Ceiling_Check* leading to a Program_Error;
- *Storage_Check* leading to a Storage_Error; Or more specifically:
  - *Pool_Check* leading to a Storage_Error;
  - *Stack_Check* leading to a Storage_Error;
  - *Heap_Check* leading to a Storage_Error;

Control of language-defined checks: The language-defined checks may be controlled by pragma or by compiler command line. The language defines the following pragmas:

- pragma Suppress: the compiler may omit the check when the result is efficient in terms of time and space optimisation; otherwise, the check is still in effect and may raise the corresponding exception. This pragma must be used with caution; i.e., should be used only if it may be proven that the presence of the check shall never fail at the level where this pragma is placed. *"Program execution is erroneous if pragma Restrictions(No_Exceptions) has been specified and the conditions arise under which a generated language-defined run-time check would fail."* (Ada RM H.4 (26)).
- pragma Unsuppress: the compiler is required to not omit the check in the region where this pragma is placed.

Exception handler: An exception handler is a specific statement part in a statement part of program unit. It aims to process an error recovery just after an exception was raised; hence the nominal process is abandoned. Such handler may be placed in the scope of the current statement part or at outer level, i.e., when the error is propagated to the caller and by transitivity until a handler is found.

Summary

- Requirements (partially) supported by Ada Dependability means: **(Req 24)**, **(Req 25)**, **(Req 26)**, **(Req 27)**, **(Req 32)**, **(Req 33)**.
- Requirements not supported by Ada Dependability means: **(Req 28)**, **(Req 29)**, **(Req 30)**.

3.1.5 Feasibility of Further Run-time Features Desired by End Users

Requirement **(Req 34)** is not addressed by Ada. Requirements **(Req 35)**, **(Req 36)**, **(Req 36)**, **(Req 37)** and **(Req 38)** come from the telecom domain, which does not use Ada. Requirement **(Req 39)** comes from the space domain, which uses Ada.

Feasibility of (Req 39)

> Run-time environments should support the possibility to disable all run-time checks.

Ada RM, 11.5 Suppressing Checks A pragma Suppress gives permission to an implementation to omit certain language-defined checks. The form of this pragmas is as follows:

```
pragma Suppress(identifier);
```

ObjectAda Raven Implementation It is possible to suppress all checks in object code (run-time checking, numeric checking). This option is equivalent to pragma Suppress on all checks.

3.2 Java/RTSJ

3.2.1 Overview of Java and the RTSJ

The *Real-Time Specification for Java (RTSJ)* [3] enhances Standard Java with features for Real-Time Programming. Although the RTSJ is provided as a set of Java APIs (in the package `javax.realtime`), the RTSJ is not just a library but requires a special realtime-capable VM. What follows is a summary of RTSJ features. The Java VM that will be used in CHESS is the JamaicaVM by aicas [5], which supports the RTSJ.

Memory management Standard Java's automatic memory management leads to unpredictability of execution time and memory needs, because Java imposes no requirements on when garbage collection takes place and for how long. For this reason, the RTSJ enhances Java with an additional manual memory management facility called *scoped memory regions*. Using scoped memory regions, one can write Java programs that do not invoke the garbage collector.

1. The RTSJ introduces the following *three kinds of dynamic memory*:
   - *Heap Memory*: memory that is subject to garbage collection.
   - *Immortal Memory*: memory that is never freed.

- *Scoped Memory*: memory where objects with a well-defined lifetime can be allocated.

2. *Management of scoped memory*: Scoped memory is further subdivided into disjoint scoped memory areas. Programmers can instruct programs to dynamically create new scoped memory areas. Dynamic creation of scoped memory areas is not limited to an initialisation phase. Dynamically created scoped memory areas can be accessed concurrently. The VM can detect in constant time when a scoped memory area becomes unreachable, and can deallocate the objects in this memory area at that point. To this end it is necessary that programmers use scoped memory areas according to a certain programming discipline. This programming discipline ensures that scoped memory areas are tree-ordered and that references across memory areas are consistent with this tree-ordering. The enforcement of this discipline requires run-time checks. Violations of these checks result in run-time exceptions.

3. *Size of scoped memory areas*: Programmers can specify the size of scoped memory areas as part of the instruction that creates a new memory area. Both an initial size and a maximum size can be specified. Programs can query the current size of a memory area at run-time.

4. *Real-time garbage collection (not part of the RTSJ)*: An alternative way of supporting predictable memory management is real-time garbage collection. The JamaicaVM supports predictable memory management, both through scoped memory areas and through real-time garbage collection. Two policies for scheduling garbage collection are supported:
   - *Work-based*: Every memory allocation is paid for with a small amount of garbage collection. The amount of garbage collection per memory allocation increases when the amount of free memory decreases. Worst-case garbage collection time per memory allocations is predictable, provided the maximum size of the heap is known beforehand.
   - *Time-based*: A fixed amount of time in each scheduling period is reserved for garbage collection. Time-based garbage collection requires that a bound on the memory allocation rate is known beforehand, and that the garbage collection time is configured accordingly.

Clocks and Time

1. The RTSJ introduces clocks with high-resolution time-types (absolute time, relative time, rational time).

2. Time values are represented by a 64-bits millisecond and a 32-bits nanosecond component.

Schedulers

1. *Abstract Scheduler class*: An abstract class provides a superclass for all scheduler implementations.

2. *The base scheduler*: Any RTSJ implementation must provide a *fixed-priority preemptive scheduler* as the base scheduler.

3. *Other schedulers*: Other scheduler implementations can be provided.

4. *Feasibility tests*: The Scheduler class has an abstract method that tests dynamically if a set of schedulable tasks is feasible. The VM invokes this method when programs dynamically create new tasks or request dynamic changes of task parameters (see below). The default implementation of the feasibility test considers any set of schedulable tasks with bounded resource requirements feasible. The feasibility test can be overridden by more useful feasibility tests.

Schedulable Objects

1. *Schedulable objects*: *Real-time threads* and *asynchronous event handlers* are instances of a common Java interface for schedulable objects. Asynchronous event handlers can be bound to both program-triggered and environment-triggered events. Schedulable objects can be created dynamically. *Periodic*, *aperiodic* and *sporadic* task activation patterns are supported. Schedulable objects have several parameters:

2. *Release parameters*:
   - *Cost*: Processing time units per release.
   - *Deadline*: The latest permissible completion time measured from the release time of the associated invocation of the schedulable object.
   - *Cost overrun handler:* An asynchronous event handler that gets invoked in case of cost overruns.
   - *Deadline miss handler*. An asynchronous event handler that gets invoked in case of deadline misses.
   - *Start time* for periodic tasks: Time at which the first release begins.
   - *Period* for periodic tasks: Interval between successive releases.
   - *Minimal inter-arrival time* for sporadic tasks: Minimal interval between successive releases.
   - *Policy for handling violation of minimal inter-arrival time* for sporadic tasks.
   - *Policy for handling overruns of the queue of outstanding releases* for aperiodic tasks.

3. *Scheduling parameters*:
   - *Priority*: The RTSJ requires a minimum range of 28 priorities.
   - *Importance*: This parameter is typically used when a system enters into a transient overload situation where it is unable to meet all deadlines. The base scheduler does not make use of this parameter.

4. *Memory parameters*:
   - *Memory budgets*: Both for immortal memory and for the initial memory area of the task.
   - *Maximum allocation rate* for Java heap.

5. *Dynamic parameter changes:* There are methods that change the parameters of schedulable objects dynamically, subject to a feasibility test for the new parameters. This feasibility test is based on the feasibility test in the Scheduler class (see above).

6. *Processing groups*: There is a mechanism to create processing groups consisting of several schedulable objects, and associate costs with processing groups. The motivation for this mechanism is to allow the execution demands of one or more aperiodic schedulable objects to be bounded so that they can be included in a feasibility analysis. Processing groups are an optional VM feature.

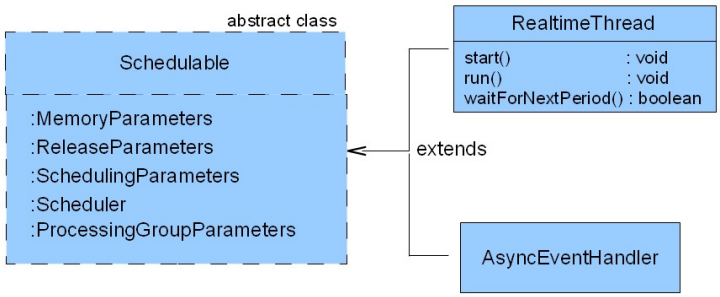


Figure 7: RTSJ: Class Hierarchy for Schedulable Objects

7. *Cost enforcement*: The RTSJ specifies cost enforcement (both for single schedulable objects and for processing groups) as an optional VM feature. The JamaicaVM currently implements cost monitoring and cost overrun detection. It does not currently implement cost enforcement (i.e., safely stopping schedulable objects that overrun their cost), but this is planned.

8. *Enforcement of memory budgets*: For the Jamaica VM, it is planned to enhance the enforcement of memory budgets to take real-time garbage collection into account.

Thread synchronisation

1. Java's main primitive for thread synchronisation are monitors.

2. The RTSJ supports both priority inheritance and the priority ceiling protocol for avoiding unbounded priority inversion. (The protocol can even be changed dynamically.)

Dynamic compilation The JamaicaVM has an optional Just-In-Time Compiler. It is configurable so that real-time tasks for which worst-case execution time is critical are not affected by it (i.e., their code is either interpreted or pre-compiled), whereas tasks for which good average time performance matters can be compiled dynamically.

3.2.2 Feasibility of Requirements for Schedulability Analysis

Feasibility of (Req 1)

> Run-time environments must provide means to implement periodic and sporadic tasks. Both program-driven and interrupt-driven sporadic tasks must be supported.

Recall from the previous section that, in the RTSJ, real-time threads and asynchronous event handlers are instances of the common superclass *Schedulable*, as depicted in Figure 7. Instances of this class are called *schedulable objects*. Periodic or sporadic tasks are represented by schedulable objects with periodic or sporadic release parameters. As shown in Figure 8, the classes *PeriodicParameters* and *SporadicParamters* extend *ReleaseParameters*. The attributes *cost* and *deadline* are common to *PeriodicParameters* and *SporadicParameters*. Additionally, *PeriodicParameters* has the attributes *period* and *start*, where the latter represents the earliest time for the initial release. The class *SporadicParameters* has an additional attribute that represents the minimal inter-arrival time.

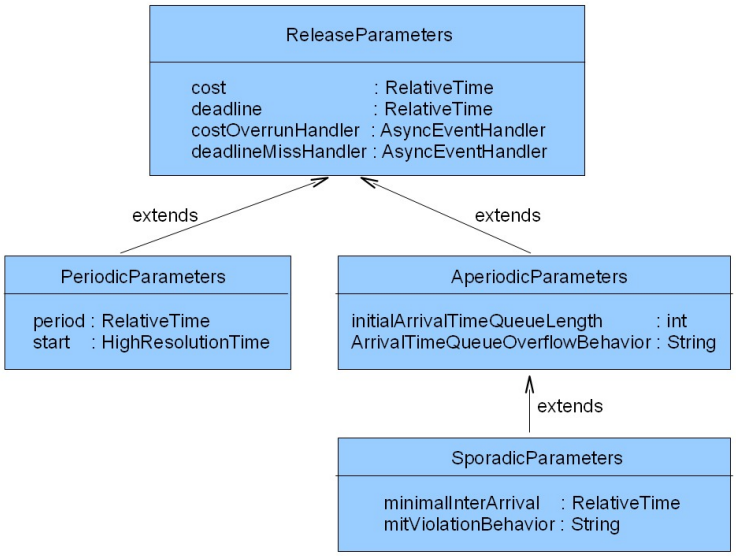


Figure 8: RTSJ: Class Hierarchy for Release Parameters

Feasibility of (Req 2)

Run-time environments must enforce periodic job releases for periodic tasks.

As explained above, periodic tasks are represented by schedulable objects with periodic release parameters. One of these release parameters is the period. Periodic tasks are implemented as real-time threads. Real-time threads have the following methods (as shown in Figure 7):

- *void start()*
- *void run()*
- *boolean waitForNextPeriod()*

The *start()*-method sets up the real-time thread's environment and then invokes its *run()*-method. If the *start()*-method was called before the earliest time for the initial task release, as specified by the *start*-attribute of *PeriodicParameters*, then the VM delays the call of *run()* until the specified *start*-time has arrived. If the thread is not in a deadline-miss condition, the method *waitForNextPeriod()* blocks until the next periodic release, as specified by the *period*-attribute of *PeriodicParameters*, and then returns *true*. If the thread is in a deadline-miss condition, the behaviour of *waitForNextPeriod()* depends on the scheduler. Figure 9 shows the typical programming pattern for periodic tasks.

Feasibility of (Req 3)

Run-time environments must enforce MIATs between job releases for sporadic tasks.

The attribute *mitViolationBehavior* of *SporadicParameters* allows the specification of different policies for enforcing MIATs. This attribute can take four possible values: the value *EXCEPT* causes an

```
class PeriodcTask extends RealtimeThread {

    PeriodicTask(PriorityParameters pri, PeriodicParameters per) {
        super(pri,per);
    }

    void run() {
        boolean noProblems = true;
        while (noProblems) {
            ... // code to be run each period
            noProblems = waitForNextPeriod();
        }
        ... // a deadline has been missed and there is no event handler
            // scheduled for recovery
    }

}
```

Figure 9: RTSJ: Implementation Pattern for Periodic Tasks

exception to be thrown if a job request arrives prior to the specified MIAT; the value *IGNORE* causes a job request to be ignored if it arrives prior to the specified MIAT; the value *SAVE* causes a job to be delayed if it arrives prior to the specified MIAT; and the value *REPLACE* causes the new job request to replace the previous job request, if the previous job has not been activated yet, and to be ignored otherwise.

Feasibility of (Req 4) and (Req 5)

> Run-time environments must provide means to implement systems where all threads (i.e., the run-time implementations of tasks) are created in a system initialisation phase, and no more threads are dynamically created thereafter.
>
> Run-time environments must provide means to implement systems where all threads do not freely terminate.

Generally, the RTSJ allows dynamic creation of real-time threads. To emulate static thread creation, programmers can implement the application's main-method such that it does nothing but starting an initialisation thread, whose task it is to create a fixed pool of threads in immortal memory, then start the first of these threads, and then terminate. Such an initialisation thread emulates static thread creation, as long as no additional thread creation statements are executed by other threads. A thread can be prevented from terminating by adding a *wait()*-statement to the end of its *run()*-method.

Additionally, the JamaicaVM allows to configure applications so that a thread pool of a fixed, but arbitrary, size is created at VM start-up. All threads that are "dynamically" created by the application are removed from this static thread pool, and returned to the pool upon thread termination. The load that results from removing or returning threads to the pool is very small compared to the load for creating new threads. So, in effect, such a configuration emulates static thread creation. Of course,

such a configuration requires a-priori knowledge of an upper bound on the number of threads that are simultaneously active.

Feasibility of (Req 6) and (Req 7)

> Run-time environments must ensure that each job has an associated worst-case execution time.
>
> Run-time environments must ensure that, for each job, the resources that are needed for executing the instruction are predictable (e.g., CPU, network and communication resources, thread-shared memory).

These are requirements on the hardware architecture and are not addressed by the RTSJ. To determine worst-case execution times and worst-case resource usage of low level instructions, one needs assumptions on the underlying hardware. For modern processors, which are highly optimised for average execution times, such assumptions are likely to be overly pessimistic, e.g., to account for infrequent cache misses. The RTSJ specifies cost monitoring and enforcement as a run-time mechanism for dealing with unlikely or rare cost overruns.

Feasibility of (Req 8)

> Run-time environments must ensure that all management tasks that operate underneath the application software (e.g., automatic garbage collection) are time-bounded.

The RTSJ supports dynamic memory allocation in scoped memory areas. The cost for allocating and deallocating such memory needs to be accounted for when determining worst-case execution times: The time for allocating scoped memory is proportional to the size of the allocated memory. The same holds for deallocation, but for deallocation the RTSJ is quite liberal on when exactly deallocation has to happen. Specifically, the RTSJ requires that the content of a scoped memory area must be deallocated between the time when the reference count for the scoped area drops to zero and the time when the next schedulable object wishes to enter the memory area. Within these bounds, the exact deallocation strategy depends on the VM. If objects that live in scoped memory areas have object finalizers, these get executed on object deallocation and their execution times must be accounted for. For competitive sharing of scoped memory areas, the RTSJ provides a *joinAndEnter()*-method that waits for the reference count of a scoped memory area to drop to zero, before entering the memory area. This statement can result in blocking, which schedulability analysis has to account for. For the sake of simplicity, it may be wise that code generators developed in CHESS omit features like object finalizers and the *joinAndEnter()*-method.

The JamaicaVM provides a real-time garbage collector, which can either run in work-based or in time-based mode. The real-time garbage collector enables the writing of predictable Java programs without having to use scoped memory areas. In the work-based garbage collection mode, a certain number of garbage collection units are executed upon each memory allocation. The exact execution time that corresponds to a garbage collection unit is processor-dependent. The exact number of garbage collection units that need to be executed per memory allocation depends on the total amount of heap space that is available: the larger heap, the smaller the required number of garbage collection

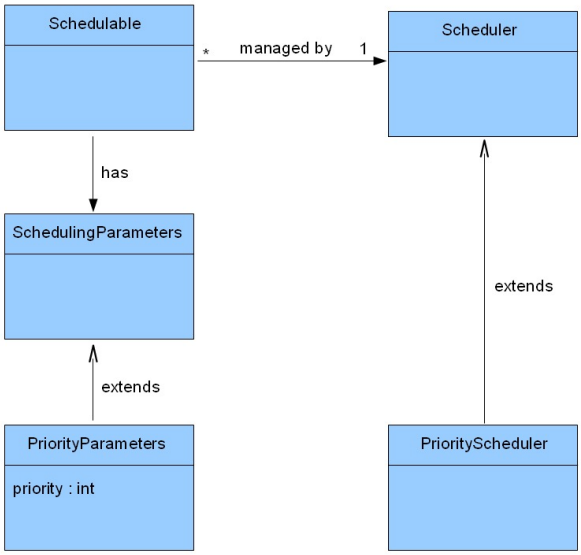


Figure 10: RTSJ: The Classes Scheduler and PriorityScheduler

units per allocation. The JamaicaVM has an associated analysis tool that executes a monitored test run, in order to determine an upper bound on the number of garbage collection units per allocation for a given heap size. Using the bound that is obtained in this way, one can compute the worst-case-garbage-collection time per allocation, as the product of this bound and the processor-dependent execution time per garbage collection unit.

Feasibility of (Req 9)

> Run-time environments must support fixed-priority preemptive scheduling. The association between threads and priorities must be static (except for dynamic priority changes due to priority inheritance).

Each schedulable object is managed by exactly one scheduler. Any RTSJ implementation must provide at least one scheduler, called the *base scheduler* and may additionally implement other schedulers. The base scheduler must be a fixed-priority preemptive scheduler with at least 28 priorities. Schedulable objects that are managed by a priority scheduler must have an associated priority represented by an integer. This situation is depicted in the class diagram in Figure 10.

Programs are allowed to modify priority parameters at any time. The base scheduler is required to not change execution eligibilities for any other reason than priority inversion avoidance or as the result of a program modifying priority parameters. In order to satisfy the requirement that priorities are not changed dynamically, it is thus important that code generators do not generate programs that dynamically modify priorities.

Feasibility of (Req 10)

> Run-time environments may optionally support earliest-deadline-first scheduling.

RTSJ implementations are not required to provide an earliest-deadline-first scheduler, but it can be implemented as an instance of the Scheduler class.

Feasibility of (Req 11) and (Req 12)

> Run-time environments must support monitors as a mechanism for synchronised access to shared memory.
>
> Monitor implementations must be such that the maximum blocking time incurred by a task is bounded and can be calculated statically.

In Java, each object is associated with an object lock that can be used to synchronise accesses to the object's state. An instance method that is decorated with the keyword *synchronized* acquires the object lock before entering the method and releases it after exiting the method. Thus, an object all of whose methods are *synchronized* is essentially a monitor.

To facilitate *conditional synchronisation*, each object has the following methods:

- *wait()*: Releases the object lock and blocks until some other thread calls *notify()* on this object.
- *notify()*: Wakes up a thread that is currently waiting on this object. The thread that is woken up is guaranteed to be one of the threads with the highest active priority among the threads that currently wait for this object (if the scheduler is the base scheduler).

Both of these methods can only be called by a thread that holds the object monitor, otherwise a run-time exception is thrown.

While Java (without RTSJ) does not specify the order in which contending threads enter a monitor, the RTSJ is very specific about how the base scheduler has to behave in this case. The general specification is quite intricate, because it accounts for various subtleties like dynamic modification of priorities and interactions of different priority inversion avoidance protocols. A corollary of the general specification is that the first thread to enter a monitor is always one with the highest active priority among the threads that wait for this monitor.

Feasibility of (Req 13)

> Run-time environments must support priority inheritance or the priority ceiling protocol, in order to bound the maximum duration of priority inversion due to synchronised access of shared resources.

The RTSJ requires the implementation of priority inheritance. Priority ceiling emulation is also specified, but its implementation is optional. The RTSJ even allows different object monitors being governed by different protocols within the same program. Furthermore, it allows dynamically changing the priority inversion avoidance protocol.

The JamaicaVM implements both priority inheritance and priority ceiling emulation.

Feasibility of (Req 14)

> Run-time environments must ensure that the overhead of interrupts is bounded.

The RTSJ does not directly address this. However, the RTSJ specifies cost monitoring and enforcement, which can be used to handle rare cost overruns due to interrupts.

Feasibility of (Req 15) and (Req 16)

> Run-time environments must support execution time monitoring in order to detect execution time overruns and deadline misses.

> Run-time environments must support the registration and invocation of programmable execution time overrun and deadline miss handlers.

Schedulable objects have release parameters that represent a deadline and a cost. Furthermore, one can install a deadline miss handler and a cost overrun handler with each schedulable object. While any VM is required to monitor deadlines, cost monitoring is an optional VM feature.

Deadline monitoring for periodic real-time threads. In case a periodic release misses its deadline and a deadline miss handler is installed for this thread, the deadline miss handler is released. Furthermore, the periodic thread is descheduled, i.e., the thread is put into a state that causes the method *waitForNextPeriod()* to block until further notice. The class *RealtimeThread* provides an instance method *schedulePeriodic()* that schedules a periodic thread that has previously been descheduled, i.e., it releases *waitForNextPeriod()* from waiting. The method *schedulePeriodic()* can be called from a deadline miss handler, and is typically called at its end.

If a periodic thread misses its deadline and no deadline miss handler is installed the method *waitForNextPeriod()* returns *false* when it is next called.

Deadline monitoring for aperiodic schedulable objects. When a deadline miss occurs, the deadline miss handler, if any, is released.

Cost enforcement. Cost enforcement is an optional VM feature. To implement cost enforcement, the VM has to measure the CPU time of a release, counting from the beginning of the release. For periodic threads, a release begins when *start()* is invoked (in case of the first release) or when *waitForNextPeriod()* returns *true*. For aperiodic threads, a release begins when *start()* is invoked (which in RTSJ 1.0.x can only happen once per thread[2]). For asynchronous event handlers, a release begins when an event triggers the handler. Cost enforcement requires that the cost overrun handler, if any, is invoked when an overrun is detected. Furthermore it requires that the schedulable object is blocked in a state *blocked-by-cost-overrun*. The transition into this blocking state has to happen within a bounded delay from the detection of the cost overrun. A blocked schedulable object becomes eligible for execution again, when a new release is triggered.

Improved cost enforcement and monitoring in RTSJ 1.1. The specification of cost enforcement in RTSJ 1.0.2 is problematic, because a VM that implements this specification is bound to sometimes block schedulable objects in inconsistent states (e.g., in states where a blocked schedulable object

[2]RTSJ 1.1 will add methods *release()* and *waitForNextRelease()* for aperiodic real-time threads. The semantics of *waitForNextRelease()* is similar to *waitForNextPeriod()*. Calling *release()* triggers a new release.

holds locks). To fix this, RTSJ 1.1 [4] will specify *cost overrun notification*, which is weaker than enforcement. Notification merely requires that the VM invokes the cost overrun handler, but does not require blocking the schedulable object. Enforcement, if any, is left to the overrun handler. Another new feature in RTSJ 1.1 are methods for querying the CPU time consumption of task releases.

Feasibility of (Req 17)

> Run-time environments must support a time-zone-independent, monotonically increasing, absolute clock. It is desirable that the granularity of time be as fine as possible.

The RTSJ requires a *system real-time clock* that is monotonically non-decreasing and measures time with respect to some epoch (e.g., 1 January 1970, 00:00:00, or system start-up time). The real-time clock must progress as uniformly and be as accurate as allowed by the underlying hardware. This implies, for instance, that it must not stall and must not be subject to leap ticks. The system real-time clock need not be synchronized with the external world. Time values are represented by a 64-bits millisecond component and a 32-bits nanosecond component.

Feasibility of (Req 18)

> Run-time environments must support the specification and enforcement of absolute delays (i.e., delays that are specified by a point in time rather than a duration relative to the point of "invoking" the delay).

The RTSJ provides a *Timer* class that can be used to this end. Timers trigger events at specified times and allow binding asynchronous event handlers to these events. The *Timer* class has two subclasses: *OneShotTimer* and *PeriodicTimer*. A one-shot timer is associated with a single release time, which can be either absolute or relative. In order to delay an action by an absolute time, one can create a one-shot timer with an absolute release time and bind to it the action. A periodic timer is associated with a single start time, which is either absolute or relative, and with a period, which is relative.

Feasibility of (Req 19)

> Generated programs must be such that each job has a single activation gate. The synchronisation and blockage of a thread that depends on other threads must be limited and statically computable.

The RTSJ does not enforce a single activation gate for releases of schedulable objects. For instance, it is allowed to write tasks of the form *if (cond0) { wait(cond1); } else { wait(cond2); }; doJob()*, where the activation gate is either the conditional wait for condition *cond1* or the conditional wait for *cond2*. So, for the RTSJ, **(Req 19)** is not enforced by the run-time environment, but it is the responsibility of code generators to produce code that adheres to this requirement.

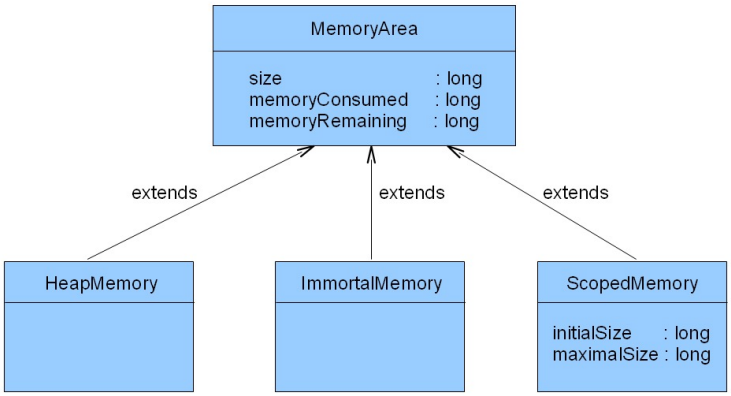


Figure 11: RTSJ: Memory areas

3.2.3 Feasibility of Requirements for Predictable Memory Consumption and Access

Feasibility of (Req 20)

> Run-time environments or generated programs should be such that it is possible to statically determine an upper bound on the amount of memory that is required to run the program.

The RTSJ does not enforce programming restrictions that are geared towards easing static prediction of memory bounds (like forbidding dynamic memory allocation, forbidding recursive methods, or forbidding dynamic memory allocation inside unbounded loops).

Feasibility of (Req 21)

> Run-time environments should monitor memory usage and enforce memory budgets (including heap, stack, pool and static memory, both code and data).

The RTSJ provides memory monitoring facilities on a per-memory-area basis and a per-schedulable-object basis.

Per memory area. As explained in Section 3.2.1, the RTSJ partitions dynamic memory into heap, immortal memory and scoped memory. This is reflected by the class hierarchy shown in Figure 11. The classes *Heap* and *ImmortalMemory* are singleton classes, whereas *ScopedMemory* can have multiple dynamically created instances. The attributes *size*, *memoryConsumed* and *memoryRemaining* of class *MemoryArea* can be queried to get information of the current size of the memory area. The attributes *initialSize* and *maximalSize* specify the initial and maximal size of a scoped memory area and have to be passed to its constructor when the scoped memory area is created. All five attributes are read-only. The VM throws an *OutOfMemoryError* when a scoped memory area exceeds its maximal size.

Per schedulable object. Each schedulable object has the following memory parameters:

- *maxImmortal*: The limit on the amount of memory the schedulable object may allocate in the immortal area. Units are in bytes.
- *maxMemoryArea*: The limit on the amount of memory the schedulable object may allocate in its initial memory area. Units are in bytes. The initial memory area can be passed to the constructor of the schedulable object.
- *allocationRate*: The limit on the rate of allocation in the heap. Units are in bytes per second.

The VM throws an *OutOfMemoryError* if any of these limits is exceeded.

Stack. Java programs throw a *StackOverflowException* if the stack memory of a thread is exceeded. Beyond this, the RTSJ does not specify monitoring of stack usage. The JamaicaVM allows to configure stack sizes for an application.

Pool. The RTSJ does not specify memory pools. Of course, memory pools for specific purposes can be implemented on top of a JVM.

Static memory. The RTSJ does not have static memory, but immortal memory comes close. The commonality between static and immortal memory is that neither of them ever gets deallocated. The difference between static and immortal memory is that the former gets entirely allocated at start-up, whereas the latter can be dynamically extended. Monitoring facilities for immortal memory have been described above.

Feasibility of (Req 22)

> Run-time environments should offer the possibility to limit or avoid the use of virtual memory.

The RTSJ does not specify that RTSJ-compliant VMs must disable or avoid virtual memory. The RTSJ offers interfaces for programmers to directly access the kinds of physical memory that a particular hardware offers.

Feasibility of (Req 23)

> Run-time environments must support atomic memory access.

Java's memory model guarantees that memory accesses are atomic for programs that are free of data races. An exception are accesses to 64-bit variables of type *long* or *double*. Accesses to such variables may require two memory accesses, and are only atomic if these variables are declared *volatile*.

3.2.4 Run-Time Mechanisms for Dependability

Java's exception mechanism Java provides a rich *exception mechanism* for identifying and classifying exceptional program conditions and errors. For faults, errors and failures that are represented as Java exception types, this mechanism can be used towards fault diagnosis and error detection, addressing requirements **(Req 24)**, **(Req 25)** and **(Req 26)**. Furthermore, Java's exception mechanism provides a way to record causal dependencies between exceptional conditions. This can be useful for tracking fault-error-failure chains, as required by **(Req 27)**.

We describe Java's exception mechanism in a bit more detail: When an exceptional condition occurs, the currently executing method interrupts its regular execution, "*throws*" an exception and *propagates* the exception to its caller. The exception is then further propagated up the call stack, unless it is caught. Exceptions can be *caught* and handled by the program. Catching an exception interrupts the exception propagation, providing the opportunity to recover from exceptional conditions. This catch-mechanism can, for instance, be used to ensure that certain exceptions do not propagate beyond the boundaries of a program component.

Exceptions have *types*. Exception types are ordered in a *subtyping hierarchy*. The top element of Java's exception type hierarchy, is the type `Throwable`. This type has the subtypes `Error`, `Exception` and `RuntimeException`. Exception types that are not subtypes of either `Error` or `RuntimeException` are called *checked exceptions*.

- `Errors` are throwables that reasonable applications should not try to catch.
- `RuntimeExceptions` are throwables that reasonable applications may try to catch. Runtime exceptions need not be declared in signatures of methods that may potentially throw these exceptions.
- *Checked exceptions* are throwables that reasonable applications may try to catch. Checked exception must be declared in signatures of methods that may potentially throw these exceptions.

Figure 12 lists some examples of built-in exceptions of these three kinds. The boundary between checked exceptions and run-time exceptions may seem a little arbitrary. The main reason for this distinction is static checkability: Verifying the absence of checked exceptions can be achieved statically by Java's type checker without producing an unacceptable number of false negatives. Run-Time exceptions, on the other hand, are beyond the scope of traditional static type checkers.

Note that `Errors` are not meant to be caught. As a result `Errors` are propagated as is, in accordance with requirement **(Req 32)**.

While Java's built-in exceptions get thrown either by the virtual machine or by Java's standard libraries, application programs can also throw exceptions. Furthermore, application programs can define their own exception types. These facilities help dealing with application-specific errors.

For identifying *error locations*, Java provides a method for printing an exception's stack trace, that is, a representation of the call stack at the time the exception was thrown (to be precise: the call stack of the thread that threw the exception).

Furthermore, Java offers programmatic support for monitoring error propagation along the so-called *catch-and-throw-again pattern*. In this pattern, methods catch an exception, do some amount of error handling, and then re-throw another exception. Exceptions can store references to other exceptions that caused them, this way supporting the recording of causal dependencies between exceptions.

Additional Java exceptions for detectable errors and failures There are number of failure conditions that get detected by all JVMs, namely those failure conditions that are represented by subtypes of Java's `Error` type as defined in Java's standard APIs. These include, for instance, `OutOfMemoryErrors` indicating that the heap memory is insufficient, `StackOverflowErrors` indicating that the stack memory is insufficient, or `InternalErrors` indicating that some unexpected internal error has occurred in the Java Virtual Machine. It is possible to define additional error types to represent other detectable failure conditions that can be recognised by a VM. For instance, the JamaicaVM detects deadlocks due to cyclic monitor entries and throws a `DeadlockError`

Errors

- `LinkageError`: Thrown to indicate that a class has some dependency on another class; however, the latter class has incompatibly changed after the compilation of the former class. Examples of subtypes:
  - `IncompatibleClassChangeError`
  - `NoClassDefFoundError`
  - `ClassFormatError`
  - `UnsatisfiedLinkError`
- `VirtualMachineError`: Thrown to indicate that the Java Virtual Machine is broken or has has run out of resources for it to continue operating. Examples of subtypes:
  - `OutOfMemoryError`
  - `StackOverflowError`
  - `InternalError`
  - `UnknownError`
- `AssertionError`: Thrown to indicate that a run-time assertion has failed.

Run-Time exceptions

- `ArithmeticException`: Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an exception of this type.
- `ClassCastException`: Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.
- `BufferOverflowException`: Thrown when a relative put operation reaches the target buffer's limit.
- `IllegalArgumentException`: Thrown to indicate that a method has been passed an illegal or inappropriate argument.
- `NullPointerException`: Thrown when an application attempts to use `null` in a case where an object is required.
- `IndexOutOfBoundsException`: Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.
- `IllegalMonitorStateException`: Thrown to indicate that a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor.
- `SecurityException`: Thrown by the security manager to indicate a security violation.

Checked exceptions

- `IOException`: Thrown to signal that an I/O exception of some sort has occurred. This type is the general type of exceptions produced by failed or interrupted I/O operations.
- `InterruptedException`: Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity.
- `ClassNotFoundException`: Thrown when an application attempts to load a non-existing class.
- `NoSuchMethodException`: Thrown when an application attempts to reflectively retrieve a reference to a non-existing method.
- `NoSuchFieldException`: Thrown when an application attempts to reflectively retrieve a reference to a non-existing field.

Figure 12: Java: Examples of exception types

when an attempted monitor entry would close a cycle. `DeadlockErrors` are not specified by Java's standard APIs, and deadlock detection is thus not supported by all JVMs. Jamaica's deadlock detection prevents certain kinds of halt failures. As another example, it would be possible to monitor the dynamic creation of threads and throw a run-time exception when threads are created after a dedicated system initialisation phase, this way addressing requirement **(Req 31)** for a static software architecture. Additional failure types could be represented as Java exception types and failure detection could be implemented, if detection is feasible and usefully complements CHESS's static dependability analyses.

Programming language safety The Java language detects and prevents a large class of errors by virtue of being a *safe programming language*. A safe programming language is *a programming language that protects its abstractions*. Every high-level programming language provides abstractions for accessing machine services. For instance, Java provides arrays as one way to access memory. Arrays are associated with operations for accessing and updating them. Safe programming languages, like Java, ensure that arrays cannot be accessed or updated in other ways, for instance by writing past the memory boundaries of other data structures. Similarly, Java ensures that objects can only be accessed and updated by indexing their fields, or that lexically scoped variables can only be accessed and updated from within their lexical scopes.

The Java language includes mechanisms that detect all behaviour that might violate Java's abstractions. These mechanisms consist of static type checking and run-time checks. Static type checking failures result in compile time errors or, in case of bytecode verification, failures to load classes. Failed run-time checks lead to exceptions being thrown. For instance, protecting the integrity of the array abstraction is primarily achieved by run-time checks that may lead to `ArrayIndexOutOfBoundsExceptions`. Protecting the integrity of object abstractions is achieved through a combination of static type checking and run-time checks that may lead to `ClassCastExceptions`.

There are, of course, errors stemming from faults at layers below the Java Virtual Machine, which a JVM will generally not detect. For instance, memory corruption due to hardware defects is likely to remain undetected by a JVM and might lead to unpredictable effects.

Isolating user applications from system services through the security manager Java's security architecture [6] provides a mechanism for isolating user applications from system resources, addressing **(Req 28)**. The core component of the Java's security architecture is the `SecurityManager`, which mediates access to system resources. System resources include, for instance, files, network resources and printers. The `SecurityManager` also mediates access to other security-critical operations, like for instance exiting the VM, stopping threads, loading native libraries, creating custom class loaders, or replacing the current security manager by another one. The Java security architecture supports *flexible access control*. That is, tailored access policies can be defined for different applications. Furthermore, the `SecurityManager` itself can be adapted by subclassing the `SecurityManager` class. The Java security architecture *separates policy expression from policy enforcement*. Policy enforcement is achieved by the `SecurityManager`. Policy expression is based on policies (as for instance defined in policy files) that map code to access permissions. The mapping depends on the following code attributes:

- Who is running the code?

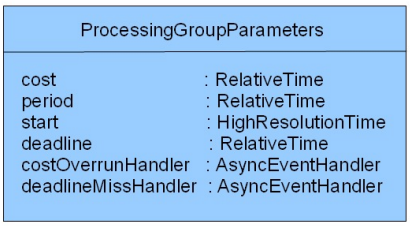


Figure 13: RTSJ: Processing group parameters

- Where does the code come from?
- Is the code digitally signed, and if so, by whom?

An empty security policy grants no permission and thus enforces execution inside a *sandbox*, achieving a maximum degree of isolation of the application from system resources.

Isolating user applications from each other through separate class loaders If the same class is loaded by different class loaders into the same instance of a JVM, the Java Language Specification guarantees that two separate copies of the same class are loaded. In other words, different class loaders give rise to different name spaces. This fact can be exploited towards achieving a certain degree of isolation between applications that run inside the same VM, which is a common scenario. This addresses requirement **(Req 28)**. For instance, VMs for the Java Micro Edition (Java ME), typically run several applications simultaneously[3]. In order to isolate Jave ME applications from each other, such VMs typically load each application with a dedicated class loader that has been created especially for running the particular instance of the particular application.

Dynamic class loading The Java platform supports *dynamic class loading*. A class can be loaded with the method `ClassLoader.loadClass(String classname)`. A class can be unloaded by deleting all references to the class, so that the class will get garbage-collected. The dynamic class loading mechanism can be used towards replacing classes in running applications, a feature that facilitates both component restarts (see requirement **(Req 29)**) and preventive and corrective maintenance as needed for instance in space to remotely update software on satellites (see Deliverable 3.3).

3.2.5 Feasibility of Further Run-time Features Desired by End Users

Feasibility of (Req 34)

> Run-time environments should support attributing resource budgets to groups of active entities (e.g., task groups or thread groups), and support monitoring/enforcing per-group budgets at run-time.

For group creation and for per-group execution time budgets, the RTSJ provides the class *ProcessingGroupParameters*, as shown in Figure 13. When multiple schedulable objects refer to the same

[3] Java ME targets embedded consumer products like mobile phones, PDAs, set-top boxes, or car infotainment systems.

instance of *ProcessingGroupParameters* (see Figure 7), they form a processing group. Each processing group has an associated *cost (budget)* and a *period*. The VM enforces that the execution time per period for the members of a processing group is bounded by the cost. The cost gets replenished every period. The motivation for this class is to allow the execution demands of one or more aperiodic schedulable objects to be bounded so that they can be included in feasibility analysis. However, periodic or sporadic schedulable objects can also be associated with a processing group.

Further attributes of a processing group are a *start* time before which none of the members of the processing group starts executing, and a *deadline* for finishing all jobs of the processing group. By default, the deadline is equal to the period. Optionally, a deadline miss handler and a cost overrun handler can be installed.

Feasibility of (Req 35)

> *Throughput.* Measured by properties such as bits/second, message/second, package/second. The analysis should consider control plane, user plane and data plane end-to-end.

These are not directly supported by the RTSJ. However, such measurements can be implemented on top of the VM by making use of the RTSJ real-time clock and Java's networking APIs. It is also possible to develop special Java APIs that offer such monitoring functionalities.

Feasibility of (Req 36) and (Req 37)

> *Execution Capacity with respect to CPU load.* For the telecom domain this can be measured in terms of number of processes, call-setup/second, signals/second, connections/second.

> *Latency.* Measured through properties such as call-setup time, message transfer, task scheduling and completion, interface specifications (services), payload. Latency is also considered as both latency to set up a connection and also latency for the data plane/payload parts.

These are not directly supported by the RTSJ. They would probably best be implemented by the underlying operating system. Java APIs that give access to such extended operating system functionalities could be developed.

Feasibility of (Req 38)

> *In Service Performance (ISP).* By enforcing system down time and Mean-Time-Between-Failures (MTBF) properties.

This is not addressed by Java or the RTSJ.

Feasibility of (Req 39)

> Run-time environments should support the possibility to disable all run-time checks.

Note that this requirement comes from the space domain, which targets Ada rather than Java.

The Java specification or the RTSJ do not allow disabling run-time checks. The JamaicaVM has a "secret" option for disabling run-time checks, but this option is not recommended. The problem with disabling run-time checks is that Java programs (including Java's built-in libraries) rely on exceptions being thrown as a result of failed run-time checks and often recover from such exceptions through exception handlers. Disabling all run-time checks is only safe if one can prove that the program (including all libraries that the program uses) never throws a run-time exception.

3.3 Real-time Linux

3.3.1 Overview of Real-time Linux

This section gives an overview of the features of Linux with respect to the adoption as a platform for real-time embedded system. The goal is to identify the relevant functionality that resides already in the standard Linux kernel 2.6, that is available through external patches or software and functionality that is missing.

Real-time properties The Linux real-time capabilities are continuously improved through the adoption of the so called `PREEMPT_RT` patch into the mainline. The patch was developed by Ingo Molnar and offers a lot of improvements in different fields.

The major improvements available through the patch are:

1. **Priority Inheritance:** To avoid unbounded priority inversion, i.e. a middle priority task preempts a high priority task for an unbounded amount of time, priority inheritance mutexes were introduced. If shared resources between a low and high priority task are protected with a PI mutex, the kernel can temporarily increase the priority of the low priority task and thus reduce priority inversion.

2. **Threaded Interrupts:** Executing interrupts in an interrupt context introduces the possibility of interrupt inversion, where high priority tasks are interrupted by the interrupt handler serving a low priority task. To avoid this effect `PREEMPT_RT` introduces threaded interrupts, making the interrupt handler itself being scheduled as a task with the associated priority. Understandably, not all interrupts can be executed within a thread context, one exception is the timer interrupt.

3. **Kernel preemption:** The Linux kernel 2.6 can be configured for different preemption modes, defining the amount of kernel code that can be preempted. Through this patch a real-time preemption mode, called `PREEMPT_RT` is available, making the largest part of the kernel code preemptable and the system behaviour more deterministic.

Linux scheduling Linux supports different scheduling policies which can be assigned on per tasks basis.

1. `SCHED_OTHER`: This is the default scheduler policy which is based on time-slices and is intended to be fair to all tasks scheduled with this policy. All processes have a fixed static priority of 0, but the dynamic priority can be adjusted (nice level). This policy is unsuitable for real-time tasks.
2. `SCHED_BATCH`: This scheduling policy is intended for batch processing. It is similar to `SCHED_OTHER`, however, the scheduler assumes that the task is CPU-intensive and therefore slightly disfavouring it in the scheduling decisions.
3. `SCHED_FIFO`: Processes scheduled with this policy have a higher static priority than all tasks scheduled with `SCHED_OTHER` or `SCHED_BATCH`, i.e. as soon as a `SCHED_FIFO` task can be scheduled, it preempts all tasks with a lower static priority. Tasks scheduled with this policy are running until they are preempted by higher static priority task, they block, or they call `sched_yield()`. This policy can be used for real-time tasks.
4. `SCHED_RR`: This policy is similar to `SCHED_FIFO` expect that they are running for a given time quantum until they are preempted and queued at the end of the list of the associated priority.

High resolution timer (`hrtimers`) A new high resolution timer subsystem was integrated in Linux 2.6.16, providing a finer resolution of timers and a better utilising of the underlying hardware. The old low resolution timer system is still present in Linux and mainly used for timeouts, such as waiting for network I/O. In contrast, the new hrtimers are used for in-kernel timings and user-space applications via `nanosleep()`, POSIX-timers and the itimer interfaces.

POSIX interfaces

1. **Memory locking** Linux supports the locking of memory, i.e. preventing that parts of the used memory are swapped out, via the `mlockall()` call.
2. **Enforcing resource limits** To some extend resources can be limited. The `setrlimit()` call allows to define soft and hard limits. The kernel enforces that soft limits for resources will not bee exceeded. The following resources can be limited:
   (a) **Virtual memory** The maximum size in bytes of the virtual memory available for a process can be limited. If a process tries to exceed the defined memory, the call fails with an `ENOMEM` error.
   (b) **CPU time** Limiting the time in seconds, a process can spend on a CPU. After reaching the soft limit, the violation is signalled to the process, allowing to react upon that. If the process continues to consume CPU time it will be terminated as soon as the hard limit is reached.
   (c) **Data segment** The data segment of a process, i.e initialised data, uninitialised data and heap, can be limited. Calls trying to exceed the limit will fail with an `ENOMEM` error.

(d) **Stack size** Limiting the maximum size of a process' stack.

(e) **Locked memory** Limits the maximum number of bytes, a process can lock his memory (`memlock()`).

(f) **Number of threads** Limits the number of processes that can be created with a given user ID.

(g) **Static priority** Defines the ceiling of a static priority a process can have.

(h) **Dynamic priority** Defines the ceiling of a dynamic priority a process can have.

(i) **Message queues** Limits the number of bytes that can be allocated for POSIX message queues.

Linux Control Groups Control groups (*cgroup*) is a framework introduced with the Kernel 2.6.24 for grouping tasks together and allowing to control their behaviour or getting arbitrary information from them. The actual functionality in terms of controlling the behaviour of tasks, is implemented in subsystems which are using the task grouping mechanism. A typical subsystem can be any kind of resource controller which accounts information through the cgroup facility. As with the Linux kernel 2.6.32 the following subsystems are implemented:

1. **Restricted access to devices:**The *devices* subsystem allows restricting the access to devices on a per cgroup basis. The system is based on white lists which explicitly grant access to devices. The devices are identified via the major / minor number mechanism, which is used by almost all devices (device drivers). One exception concerns however network interfaces.

2. **Partition the system based on CPUs:** On multiprocessor machines *cpuset* can partition the system based on CPUs, allowing for example to separate real-time and non real-time applications on different CPUs.

3. **Restrict memory usage:** The *mem_cgroup* subsystem allows limiting memory usage for tasks in a cgroup. It accounts anonymous, page cache (mapped and unmapped) and swap cache memory pages. This subsystem however is not set to limit stack usage.

4. **Limiting CPU bandwidth:** The *cpu_cgroup* allows defining the minimum amount of CPU bandwidth available for groups of tasks.

5. **CPU accounting:** With *cpuacct* the consumed CPU time among all threads within a cgroup can be identified. The time is measured in nanoseconds.

6. **Stopping / Starting groups of tasks:** With the *freezer* subsystem all tasks in a cgroup can be stopped or started.

7. **Namespaces:** The *ns* subsystem allows defining namespaces for certain entities such as PIDs. Usually PIDs are unique on a system but with the *ns* subsystem they can be defined unique for one cgroup. The available namespaces are:

   - **PID:** Process IDs are only unique within a cgroup.
   - **UTS:** The name and information about the kernel as retrieved via the system call `uname()` can be define per cgroup.
   - **USER:** Allows to define different user sets per cgroup.

- **NET:** Defines different views for processes within a cgroup on the network stack, network interfaces, routing table, firewall rule, etc.
- **IPC:** Unshare IPCs and have a private set of IPC objects inside a cgroup.

8. **Generic resource controller:** The Linux kernel 2.6 offers a generic resource controller framework allowing to add new resource controllers which are currently not implemented. Functionality that is identified to be missing in the Linux kernel could probably be implemented by utilising this framework.

3.3.2 Feasibility of Requirements for Schedulability Analysis

Feasibility of (Req 1), (Req 2), (Req 3)

> Run-time environments must provide means to implement periodic and sporadic tasks. Both program-driven and interrupt-driven sporadic tasks must be supported.
>
> Run-time environments must enforce periodic job releases for periodic tasks.
>
> Run-time environments must enforce MIATs between job releases for sporadic tasks.

The Linux kernel 2.6 provides no specific support for periodic and sporadic tasks. However, periodic tasks behaviour can be achieved by calling *clock_nanosleep()*, while sporadic tasks can be created using the signal handler from the POSIX interfaces.

The function *clock_nanosleep()* allows the caller to sleep for a defined time period. The period can either be measured against system-wide real-time clock by passing the option `CLOCK_REALTIME` to the function or it can be measured against a monotonically increasing clock by passing `CLOCK_MONOTONIC`.

The time specification structure can be interpreted as being relative or absolute. By passing the flag `TIMER_ABSTIME`, the time specification is absolute. This prevents the common problem of timer drift when using a regular sleep function like *sleep()* or *nanosleep()*.

Periodic task implementation *Sleep()* or *nanosleep()* calls allows to suspend a given task for a given amount of time, however they do not allow to work with absolute time. Figure 14 illustrates a serious problem that can arise while trying to manage periodic activation through relative time: if the red periodic task is preempted by another process after calculating the sleeping interval and before calling *sleep()* or *nanosleep()* then a wrong sleeping interval is calculated and set. The task is woken up later than desired which yields to a period miss.

A possible solution to this problem can be obtained by using *clock_nanosleep()* function together with the `TIMER_ABSTIME` flag to designate absolute time usage; so doing it does not matter if the task gets preempted between the time it takes a timestamp and the sleep function because it arms the timer to expire at a specific time in the future.

Figure 15 shows an example implementation of a periodic task using monotonic clock (`CLOCK_MONOTONIC`).

By enabling real-time preemption and definition of priority rules it is ensured that the kernel and other user-space tasks don't get in the way of the processing that needs to be done by high priority periodic task.

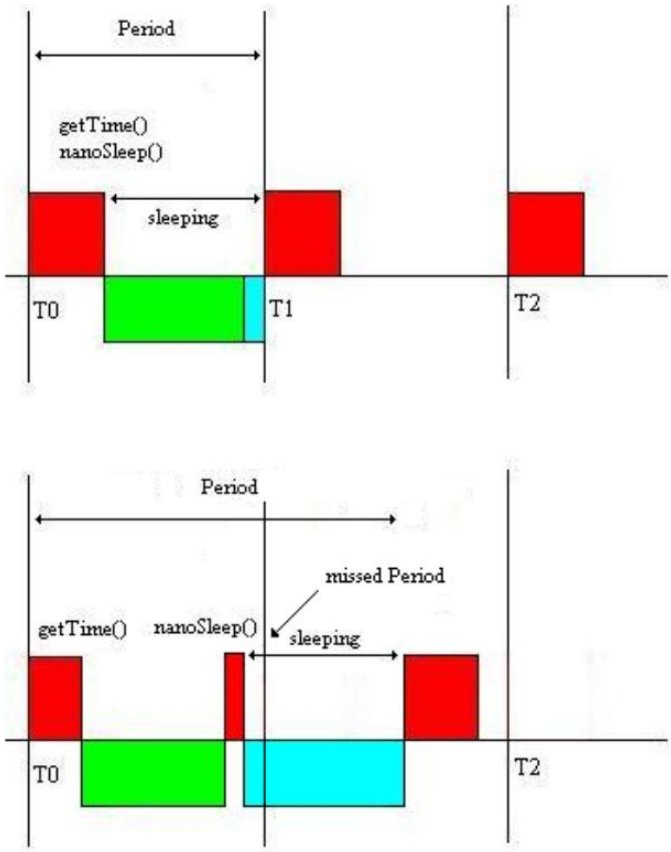


Figure 14: Period miss when relying on relative time.

Jitter is only the variable deviation from ideal timing event. Scheduling jitter is the delay between the time when task shall be started, and the time when the task is being started (see figure 16). Jitters can result from physical phenomena in hardware (noise), from interrupt latency, from concurrent task processing and so on. Use of `PREEMPT_RT` minimises jitter.

Sporadic task implementation A periodic server is a mechanism for scheduling an aperiodic workload in a way that is compatible with schedulability analysis techniques originally developed for periodic task systems. Aperiodic requests (jobs) are placed in a dedicated queue upon arrival.

The server activates at times $t_1, t_2, \ldots$ such that $t_{i+1} - t_i = T_s$, where T_s is the nominal server period, and executes at each activation for up to C_s, where C_s is the server budget. If the server uses its budget then it is preempted and its execution is suspended until the next period.

If the server is scheduled to activate at time t and finds no queued work, it is deactivated until $t + T_s$. In this way the aperiodic workload is executed in periodic bursts of activity; i.e., its execution is indistinguishable from a periodic task

Feasibility of (Req 4)

> Run-time environments must provide means to implement systems where all threads (i.e., the run-time implementations of tasks) are created in a system initialisation phase, and no more threads are dynamically created thereafter.

```
periodic_task() {

    clock_gettime(CLOCK_MONOTONIC, time);
    time = time + period;

    while(1) {
        clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME,time, NULL);
        do_work();
        time = time + period;
    }
}
```

Figure 15: An example for a periodic thread implementation.

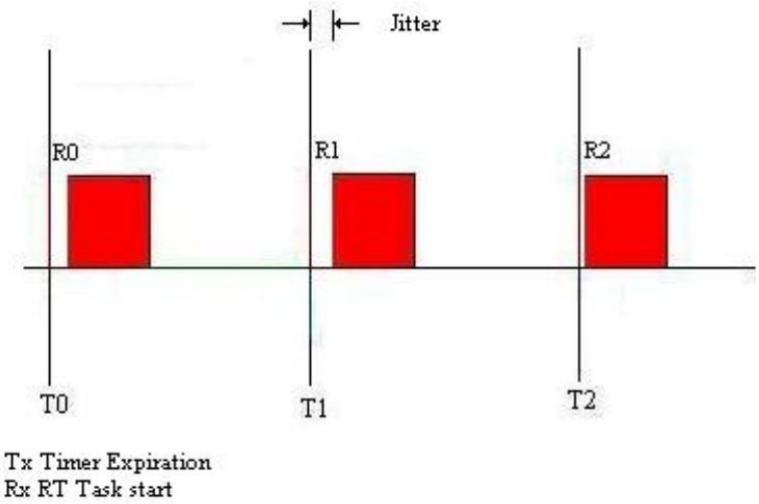


Figure 16: Occurrence of jitter in real-time tasks.

The Linux kernel 2.6 allows dynamic thread creation. In order to limit the creation of thread to an *system initialisation phase* manual work has to be done. For example a dedicated thread could be selected to spawn all required threads of an application during startup. During this time the initialisation code for a real-time tasks initialises the real-time task structure and informs Linux kernel of period.

Prohibiting further thread creation after this point can not be maintained by the Linux kernel itself. Possible solutions could be to define cgroups which prevents creating further threads on an application level. The other solution is to move this responsibility to the code generator, which assures that no further threads are created.

Further investigations upon run-time environment are required in order to prove that the dynamic creation of other real-time task on the same kernel (e.g. performed by other applications) can be prevented.

Feasibility of (Req 5)

```
void *thread_main(void* arg)
{
        // Do stuff
}

void *thread_main_wrapper(void* arg)
{
        pthread_cleanup_push(exit_handler, NULL);
        thread_main(arg);
        pthread_cleanup_pop(1);
}
```

Figure 17: Recommended approach to detect thread termination.

> Run-time environments must provide means to implement systems where all threads do not freely terminate.

Utilising the POSIX API there can be different situations leading to thread termination:

- The thread main function returns.
- The thread calls *pthread_exit()*.
- The thread is cancelled by an other thread with the call *pthread_cancel()*.
- The thread terminates the whole process via *exit()*.

It is not possible to prohibit calls to these functions, however handlers could be installed to detect thread termination. The termination caused by calling *pthread_exit()* or *pthread_cancel()* can be detected by installing cleanup handler. This can be achieved with the call of *pthread_cleanup_push()* and *pthread_cleanup_pop()* at the beginning and at the end of the thread's main function. However it does not cover the case where a thread terminates itself by calling *return*. To cover this case it is recommended to wrap the thread's main function in an other function that contains the calls of *pthread_cleanup_push()* and *pthread_cleanup_pop()* as shown in Figure 17.

This approach does not detect process termination via the call *exit()*, however this can be achieved with registering *atexit()* or *on_exit()* handlers.

Feasibility of (Req 6), (Req 7) and (Req 8)

> Run-time environments must ensure that each job has an associated worst-case execution time.
>
> Run-time environments must ensure that, for each job, the resources that are needed for executing the instruction are predictable (e.g., CPU, network and communication resources, thread-shared memory).
>
> Run-time environments must ensure that all management tasks that operate underneath the application software (e.g., automatic garbage collection) are time-bounded.

The Linux kernel 2.6 has an additional configuration option, `CONFIG_PREEMPT_RT`, which causes all kernel code outside of spinlock-protected regions and interrupt handlers to be eligible for non-voluntary preemption by higher priority kernel threads. With this option, worst case latency drops to (around) single digit milliseconds, although some device drivers can have interrupt handlers that will introduce latency much more worse than that. If a real-time Linux application requires latencies smaller than single-digit milliseconds, use of the `CONFIG_PREEMPT_RT` patch is highly recommended. The patch converts Linux into a fully preemptible kernel.

Feasibility of (Req 9)

> Run-time environments must support fixed-priority preemptive scheduling. The association between threads and priorities must be static (except for dynamic priority changes due to priority inheritance).

It is possible to define the real-time scheduling policy and priority of a thread by using the *sched_setscheduler()* and *sched_setparam()* API.

sched_scheduler() The following real-time policies are supported:

- `SCHED_FIFO`: A first-in, first-out policy; and
- `SCHED_RR`: A round-robin policy.

They are intended for time-critical applications that need precise control over the way in which runnable processes are selected for execution.

Processes scheduled under one of the real-time policies (`SCHED_FIFO`, `SCHED_RR`) have a *sched_priority* value in the range 1 (low) to 99 (high). As the numbers imply, real-time processes always have higher priority than normal processes.

Conceptually, the scheduler maintains a list of runnable processes for each possible *sched_priority* value. In order to determine which process runs next, the scheduler looks for the non-empty list with the highest static priority and selects the process at the head of this list.

sched_setparam() *sched_setparam()* checks the validity of param for the scheduling policy of the process. The value *param->sched_priority* must lie within the range given by *sched_get_priority_min()* and *sched_get_priority_max()*.

Feasibility of (Req 10)

> Run-time environments may optionally support earliest-deadline-first scheduling.

Not Supported. It is possible to be implemented in the next releases (RTAI supports this feature also). There is also a external project which implemented a EDF scheduling policy called `SCHED_DEADLINE`.

Feasibility of (Req 11)

> Run-time environments must support monitors as a mechanism for synchronised access to shared memory.

It is supported by Linux kernel 2.6 using `CONFIG_PREEMPT_RT` and `PTHREAD_PRIO_INHERIT` mutex.

In NPTL, thread synchronisation primitives (mutexes, thread joining, etc.) are implemented using the *futex()* (short for "fast user space mutex") system call. The *futex()* system call provides a method for a program to wait for a value at a given address, and a method to wake up anyone waiting on a particular address. This system calls are typically used to implement the contended case of a lock in shared memory (`FUTEX_WAKE`, `FUTEX_WAIT`).

Feasibility of (Req 12) and (Req 13)

> Monitor implementations must be such that the maximum blocking time incurred by a task is bounded and can be calculated statically.
>
> Run-time environments must support priority inheritance or the priority ceiling protocol, in order to bound the maximum duration of priority inversion due to synchronised access of shared resources.

It status of the Linux kernel 2.6 with `PREEMPT_RT` patch is:

1. Priority inheritance supported
2. Priority ceiling not supported

The *pthread_cond_timedwait()* function allows an application to give up waiting for a particular condition after a given amount of time.

RT-mutexes (supported by Linux kernel 2.6) with priority inheritance are used to support PI-futexes, which enable *pthread_mutex_t* priority inheritance attributes (`PTHREAD_PRIO_INHERIT`).

Feasibility of (Req 14) and (Req 15)

> Run-time environments must ensure that the overhead of interrupts is bounded.
>
> Run-time environments must support execution time monitoring in order to detect execution time overruns and deadline misses.

It is possible to use the function *ftrace* to detect interrupt and scheduling latencies.

Feasibility of (Req 16)

> Run-time environments must support the registration and invocation of programmable execution time overrun and deadline miss handlers.

It is possible to set alarm attached to handler to check execution time overrun and deadline miss.

Feasibility of (Req 17) and (Req 18)

> Run-time environments must support a time-zone-independent, monotonically increasing, absolute clock. It is desirable that the granularity of time be as fine as possible.
>
> Run-time environments must support the specification and enforcement of absolute delays (i.e., delays that are specified by a point in time rather than a duration relative to the point of "invoking" the delay).

Linux supports both time-zone-dependent and monotonically increasing time-zone-independent timers. They can be used with the high resolution timers which are supported by Linux kernel 2.6 and the `PREEMPT_RT` patch (with the option `CONFIG_HIGH_RES_TIMERS=y`). The monotonically increasing timer can be used with the argument `CLOCK_MONOTONIC` in the call *clock_nanosleep()* which also supports relative and absolute timings.

Absolute and relative timings can be configured with the `HRTIMER_MODE_REL` and `HRTIMER_MODE_ABS` option.

Feasibility of (Req 19)

> Generated programs must be such that each job has a single activation gate. The synchronisation and blockage of a thread that depends on other threads must be limited and statically computable.

The *pthread_cond_wait()* function allows an application to give up waiting for a particular condition.

3.3.3 Feasibility of Requirements for Predictable Memory Consumption and Access

Feasibility of (Req 20), (Req 21)

> Run-time environments or generated programs should be such that it is possible to statically determine an upper bound on the amount of memory that is required to run the program.
>
> Run-time environments should monitor memory usage and enforce memory budgets (including heap, stack, pool and static memory, both code and data).

The function *getrusage* returns resource usage measures according to the following input parameters:

- RUSAGE_SELF Returns resource usage statistics for the calling process, which is the sum of resources used by all threads in the process.
- RUSAGE_CHILDREN Returns resource usage statistics for all children of the calling process that have terminated and been waited for. These statistics will include the resources used by grandchildren, and further removed descendants, if all of the intervening descendants waited on their terminated children.
- RUSAGE_THREAD (since Linux 2.6.26) Return resource usage statistics for the calling thread.

The resource usages are returned in the structure pointed to by usage, which has the following form:

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims (soft page faults) */
    long   ru_majflt;        /* page faults (hard page faults) */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* IPC messages sent */
    long   ru_msgrcv;        /* IPC messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```

Feasibility of (Req 22)

> Run-time environments should offer the possibility to limit or avoid the use of virtual memory.

Real-time task can not tolerate the unpredictable overhead by virtual memory and paging, so the standard POSIX requires that those features could be disabled.

For that purpose, Linux implements *mlock()* and *mlockall()* system calls.

Feasibility of (Req 23)

> Run-time environments must support atomic memory access.

Linux supports atomic memory accesses through an internal atomic api. The implementation heavily depends on the underlying processor architecture with the simplest one just disables the interrupts temporarily. Function like *pthread_mutex_lock()* obviously depends on that implementation.

3.3.4 Run-Time Mechanisms for Dependability

Feasibility of (Req 24), (Req 25), (Req 26), (Req 29), (Req 30)

> Run-time environments should provide support for *fault diagnosis* mechanisms that identify and record the cause(s) of error(s) in terms of both location and type, and possibly notify the presence of an unexpected fault (i.e., not considered at model-level). E.g.: alpha-count to discriminate between transient and permanent faults.

> Run-time environments should provide support for *error detection* techniques that identify the presence of an error, and possibly notify the presence of an unexpected error (i.e., not considered at model-level). E.g.: parity check to detect 1 bit flip errors.
>
> Run-time environments should provide support for *failure detection* techniques that identify the presence of a failure mode, and possibly notify the presence of an unexpected failure mode (i.e., not considered at model-level). E.g.: failure detectors to check halt failures.
>
> Run-time environments should provide support for implementing *restarting mechanisms* to repair/replace specific (hw/sw) components. E.g.: restart for sw rejuvenation.
>
> Run-time environments should provide support for *reconfiguration actions* through mechanisms that allow either switches in spare components or reassign tasks among non failed components.

The threats diagnosis and detection is dependent on the concrete application needs. Linux supports watchdog functionality via hardware watchdogs but also via a software implementation. A watchdog daemon can control the shutdown or reboot behaviour and can also perform tests to determine the system status. Examples of these tests are:

- Exceeding a defined maximum load threshold
- Minimal amount of free virtual memory
- Exceeding a maximum temperature threshold
- Test if network interfaces receive traffic

Furthermore, user defined tests can be added by supplying a test binary. In the case of a failure, repair actions can be performed. The default shutdown or reboot action can be extended by defining a repair binary, which performs the repair action.

Feasibility of (Req 27)

> Run-time environments should provide support to monitor the conformance of the *fault-error-failure chains* perceived during operation with respect to the fault-error-failure chains defined at model-level, possibly raising a notification.

The Linux kernel 2.6 has no explicit support to perform monitoring of fault-error-failure chains. This functionality can be added via an additional user defined component which can rise notifications. However, there could be issues if the watchdog daemon is planned to be used.

Feasibility of (Req 28)

> Run-time environments should provide support for mechanisms improving the *isolation* between non interacting components (i.e., components not having functional relations). E.g.: adoption of MMU.

Linux supports the utilisation of memory management unit (MMU) on CPUs to protect the memory regions of processes from invalid access. With the help of a MMU, space isolation for software components can be achieved. Time isolation can be achieved by using a appropriate scheduler.

3.3.5 Feasibility of Further Run-time Features Desired by End Users

Feasibility of (Req 34)

> Run-time environments should support attributing resource budgets to groups of active entities (e.g., task groups or thread groups), and support monitoring/enforcing per-group budgets at run-time.

The functions *getrlimit()* and *setrlimit()* gets and sets resource limits respectively. Each resource has an associated soft and hard limit, as defined by the *rlimit* structure.

Some resources are:

- `RLIMIT_AS` The maximum size of the process's virtual memory (address space)
- `RLIMIT_MEMLOCK` The maximum number of bytes of memory that may be locked into RAM.
- `RLIMIT_STACK` The maximum size of the process stack

Feasibility of (Req 35), (Req 36), (Req 37), (Req 38), (Req 39)

> *Throughput.* Measured by properties such as bits/second, message/second, package/second. The analysis should consider control plane, user plane and data plane end-to-end.
>
> *Execution Capacity with respect to CPU load.* For the telecom domain this can be measured in terms of number of processes, call-setup/second, signals/second, connections/second.
>
> *Latency.* Measured through properties such as call-setup time, message transfer, task scheduling and completion, interface specifications (services), payload. Latency is also considered as both latency to set up a connection and also latency for the data plane/payload parts.
>
> *In Service Performance (ISP).* By enforcing system down time and Mean-Time-Between-Failures (MTBF) properties.
>
> Run-time environments should support the possibility to disable all run-time checks.

It is possible to use the *ftrace* system call based upon the *frysk* project that launches the Frysk Execution Analysis Tool allowing to monitor running processes and threads (including creation and destruction events), monitor the use of locking primitives, expose deadlocks, gather data and debug any given process.

3.4 OSE

3.4.1 Overview of OSE

OSE is a distributed real-time priority based operating system developed by ENEA that features task switching and pre-emptive scheduling and uses the principle of message passing between processes. In OSE developers can split designs into independent and transparent functional blocks with their memory pools and protection mechanisms. OSE does not rely on semaphores and shared memory (it provides the features though) for communication because these mechanisms make it harder

to track and ensure independence if a failure occurs in high availability systems. It offers the concept of process blocks which can be used to group and form logical blocks of processes. In an OSE system, there is one configuration file and one error handler per kernel. So the error handler is not fragmented over the application and thus it can be upgraded on the fly without any change in the application code. These features in OSE help to provide some level of separation of concerns. Figure 18 shows the internal structure of OSE.

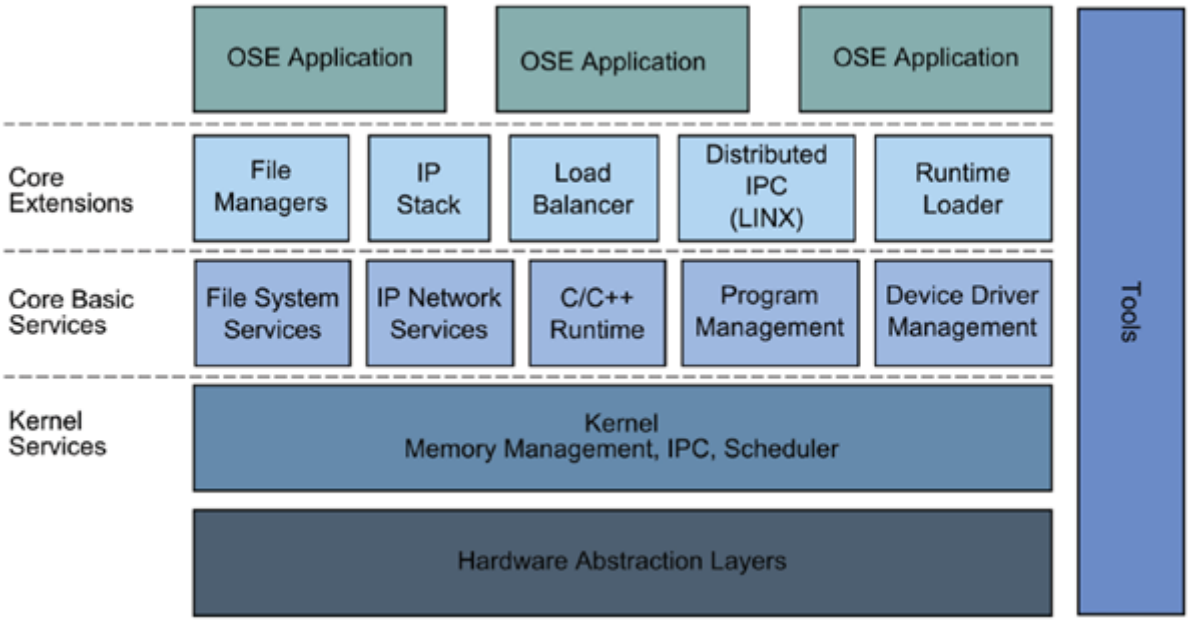


Figure 18: OSE Internal Structure

OSE comes in different editions: OSE Delta (Multicore RTOS), OSEck (Compact Kernel, DSP-optimised version of OSE RTOS) and OSE Epsilon (Optimised Microcontroller RTOS).

A categorised explanation of OSE features relevant for CHESS is offered as follows:

- **Language Support:** OSE provides C run-time library (CRT) support for the main part of the ANSI/ISO C standard definition. OSE also provides run-time support for a number of C++ features (such as exceptions and templates), but in the case of C++, the run-time libraries are delivered together with the compiler. The level of C++ support also depends on how well the compiler environment has been adapted to the OSE run-time libraries. OSE has official C/C++ support for the GCC compiler for all targets (including SFK) and for the ARM RVCT compiler. The latest OSE version and supported targets and compiler versions are detailed in the OSE Release Information, but typically a current version of the GCC compiler is supported for all targets (for the ARM targets an ARM RVCT compiler is also offered).

- **Hardware Platforms:** OSE5 Delta is available for Xscale, ARM, PPC and MIPS32 architectures. The following processor families are supported: ARM9E, ARM11, Xscale, Cavium Octeon Plus (CM58xx and CM56xx), IBM PPC4xx, IBM PPC7xx, Freescale MPC86xx (multicore support on MPC8641D), Freescale MPC85xx (multicore support on MPC8572DS, QorIQ P2020), Freescale MPC83xx, Freescale MPC82xx, Freescale MPC74xx, MIPS32

OSEck supports LSI logic (SP26xx, DSP 16K ZSP500 ZSP400 and DSP cores) CEVA (X16xx, TeaKLite-III), Texas Instruments C6000 and C5000 family, Freescale MSC81xx and MPC5xx series and Analog Devices' TigerSHARC TS201S, TS101S.

- **Inter-process Communication:** OSE offers Link Handlers (LNH) and its new version, ENEA LINX, which provides reliable, high-performance, inter-process communications services that make complex distributed systems easier to conceptualise, model, partition, and scale. It uses direct message passing and supports any distributed system topology, from a single processor on a single blade, to large networks with complex cluster topologies deployed on hundreds of processors in a multi-rack system. It enables different processes running on different boards to communicate with each other. Link handlers are typically used to make the processes on the remote side of a communication link appear as local processes. This enables the ordinary OSE communication primitives to be used with remote processes, just as if they were local. So it provides location transparency and scalability which are required in telecom systems.

- **Upgrade:** By using Link Handlers, OSE is capable of dynamic discovery of connections which enables the addition of boards to the system during run-time. OSE Program Handlers, on the other hand, enable run-time update and upgrade of software modules which is very important for the up-time of the systems.

- **Logging & Debugging:** A logging mechanism provides traceability in the system and can be used for debugging and system monitoring. The RAMLog feature in OSE provides the mechanism to log information in order to make system analysis and debugging easier. This log can include information such as kernel and process states. For debugging, ENEA offers a tool called ENEA OPTIMA (proprietary tool) which is a suite of powerful system debug and profiling tools for the Eclipse environment and the Enea OSE real-time operating system. Optima features a complete integrated development environment, including the Eclipse platform and Eclipse C/C++ development tools. OPTIMA has the following features:
  C/C++ IDE, C/C++ Debug Support (based on GDB), System Browser, Pool Browser, Post-Mortem Dump Support, System Profiler (Collects and shows statistics such as total CPU load, CPU load per process, CPU load per priority level, and heap usage per process in OSE applications. Also shows statistics collected from counters instrumented into the source code.), Log Manager (import/export and visualisation of logs supporting XML)

- **Security:** For communication, OSE provides security protocols such as IPSec. OSE also supports different memory domains, which helps with memory protection and maintaining the integrity of different parts of the system. It provides access permission controls for different regions and domains. This way OSE protects critical parts of the system such as kernel and special memory regions against malicious application actions which leads to better robustness in the system.

- **File System:** OSE provides some file system options such as Journaling Extensible File system Format (JEFF), ImageTool and FlashFX. JEFF is a POSIX.1 compliant format, designed for performance and high availability systems. It provides crash-safe and transactional update features.

- **Memory Management:** The Memory Manager in OSE provides support to dynamically create shared virtual memory, as well as to map external memory into the virtual space, for example memory shared with a Linux environment. While providing support for "bare-metal" performance on traffic cores, one or a few cores can run higher layers of networking protocol control software as well as operation and maintenance (O&M) software without interfering with the traffic cores. Also as mentioned earlier, memory management in OSE provides the concept of memory domains and access protection.

3.4.2 Feasibility of Requirements for Schedulability Analysis

OSE prototype for CHESS The scheduler in OSE uses round robin priority-based pre-emptive approach for scheduling tasks. However, as a CHESS prototype, it is possible to implement and add a more appropriate scheduler for the needs of CHESS. Such a scheduler is somewhat available in OSEck. The feasibility of the requirements in this section has been evaluated considering such a prototype. The motivation behind developing such a prototype is that, for example, definitions of periodic tasks in the form of period and execution time values are not explicitly supported and should be implemented through timer interrupts.

(Req 1) *Representing periodic and sporadic tasks:* It is possible to implement this feature using hardware timers for generating periodic interrupts and invoking an interrupt process representing a task. Kernel timeout mechanisms can also be used for signaling a process periodically.

(Req 2) *Enforcing periodic job release:* The main scheduling principle in OSE is priority based pre-emptive scheduling. From OSE version 5.x, it is possible to enable a time slicing feature for prioritised processes. The feature is enabled globally but the actual time slice value is set per process. Background processes use only time slicing and cannot be assigned a priority. This feature can be implemented in CHESS prototype.

(Req 3) *Enforcing minimum inter-arrival times:* In OSE, if a process is awakened and has highest priority of ready processes, it will be run by the kernel, regardless of when it ran previously. There is no state or history in the kernel for this. However, it is possible to implement this feature such that when a task is released, it is checked to see whether it arrives before its specified MIAT or not, and if so its execution will be delayed (e.g. using delay() system call). So although this feature is not directly supported in OSE, it is implementable.

(Req 4) *Creating all threads in system initialisation phase and no dynamic thread creation thereafter:* It is possible to define all processes in a system initialisation phase, and not use any instructions which create dynamic threads/processes in their implementation code.

(Req 5) *No thread termination:* A process that is defined as a static process at compile time will not be allowed to be killed. So this requirement is implementable using static processes.

(Req 6), **(Req 7)** *Worst-case execution time and resource usage for each job:* It is not supported in OSE. Can be implemented as an OSE prototype for CHESS.

(Req 8) *Ensuring that execution times for resource management are time-bounded:* It depends on what resources we are talking about. For example, allocating a memory buffer from a pool using the alloc() system call is fast and deterministic. The memory pool has no merge/split or defragmentation/garbage collection.

(Req 9) *Fixed-priority scheduling:* Yes

(Req 10) *Earliest-deadline first scheduling:* Not feasible in OSE but can be implemented as part of a CHESS prototype.

(Req 11) *Support Monitors:* OSE provides mutexes and Monitor behaviour can be implemented using them.

(Req 13) *Supporting priority inheritance or the priority ceiling protocol:* In OSE version 5.x, a priority-inheriting mutex function has been added that can temporarily raise its priority when needed. However, we recommend another design approach using processes and message passing to avoid these error prone situations. OSE supports attributes on mutexes such as OSE_MUTEX_PRIO_INHERIT to enable priority inheritance.

(Req 15) *Execution time monitoring to detect execution time overruns and deadline misses on a per-job basis:* There is no direct support for it. However it is possible to implement it using other mechanisms but it incurs lots of overhead and penalty. For example, a high prioritised process can send a signal to detect this. We can also implement it using swap-in and swap-out handlers that are triggered when a process starts and stops executing.

(Req 16) *Registration and invocation of programmable violation handlers (for handling execution time overruns, deadline misses and memory budget violations):* There is only support for memory budget violation handlers. Errors can be caught on different levels, process level or system level (memory violation is an error which we tend to see more often than others in telecommunication systems). Other aspects not explicitly supported, can be implemented using swap-in and swap-out.

(Req 17) *Time-zone independent monotonically increasing, absolute clock:* As an operating system, OSE relies on the clock provided by the hardware. It can use clock chips to provide a monotonically increasing time-zone independent clock for applications.

(Req 18) *Absolute delays:* OSE provides APIs for this requirement.

(Req 19) *Single activation gate:* This is an implementation requirement. Processes can be implemented this way.

3.4.3 Feasibility of Requirements for Predictable Memory Consumption and Access

(Req 20) *Upper bounds on memory usage of tasks:* This is possible as a defined memory space is allocated to processes when they are specified.

(Req 21) *Monitor memory usage and enforce memory budgets:* This is already feasible in OSE.

(Req 22) *Avoid or limit virtual memory:* If needed, it is possible in OSE to define memory only in physical memory areas.

(Req 23) *Atomic memory access:* This can be implemented using semaphores.

3.4.4 Run-Time Mechanisms for Dependability

In this section, we describe OSE mechanisms that help with building a dependable and secure systems targeting the requirements listed in Section 2.3.

- OSE has a dedicated component in its core architecture for information logging called RAMLOG. It facilitates analysis and debugging. RAMLOG is available even in sensitive contexts such as real-time sensitive code, interrupt handling code, error situations when working from

error handlers or when everything else has failed. RAMLOG basically offers a central isolated place for storing errors and other types of necessary information. Using RAMLOG service helps with isolation between faulty components and the mechanism for documenting the situations leading to that fault. So the logging component is actually protected from any problems in the actual component using this service.

- Run-mode Monitor: (Actually a debugging mechanism) The Run-Mode Monitor (RMM) is an OSE Process (ose_monitor) that allows system and source code run-mode debugging of an OSE system. Using the RMM it is e.g. possible to set breakpoints, step through source code, and access internal OSE structures. The RMM also allows CPU profiling, user value profiling, access to the target memory, direct control of processes, and monitoring of real-time events using event actions.
- OSE Monitor Protocol: The OSE monitor protocol is a signal protocol for communication between a debugger (client) and the RMM. The fact that it is a signal protocol makes it possible to debug any OSE system that is reachable through link handlers or OSE gateways.
- Through the concept of *load modules* which are dynamically loadable programs and are analogous to Windows .exe files, OSE allows dynamic loading of programs. This feature is important for hot-swap and hot-plugging features and restarting different parts of the system (enabling restart at different granularities).
- As mentioned in previous sections in introduction of OSE, OSE has a unique way to detect and handle errors. Instead of the common approach to have a error treatment mechanism (e.g. try-catch) after each block/statement, OSE allows the user to have to centralise treatment of errors. This mechanism is applicable at three levels: process, block and system level. It is also possible for applications to call the error handlers if needed. These error levels and how errors are propagated in the system is shown in figure 19.

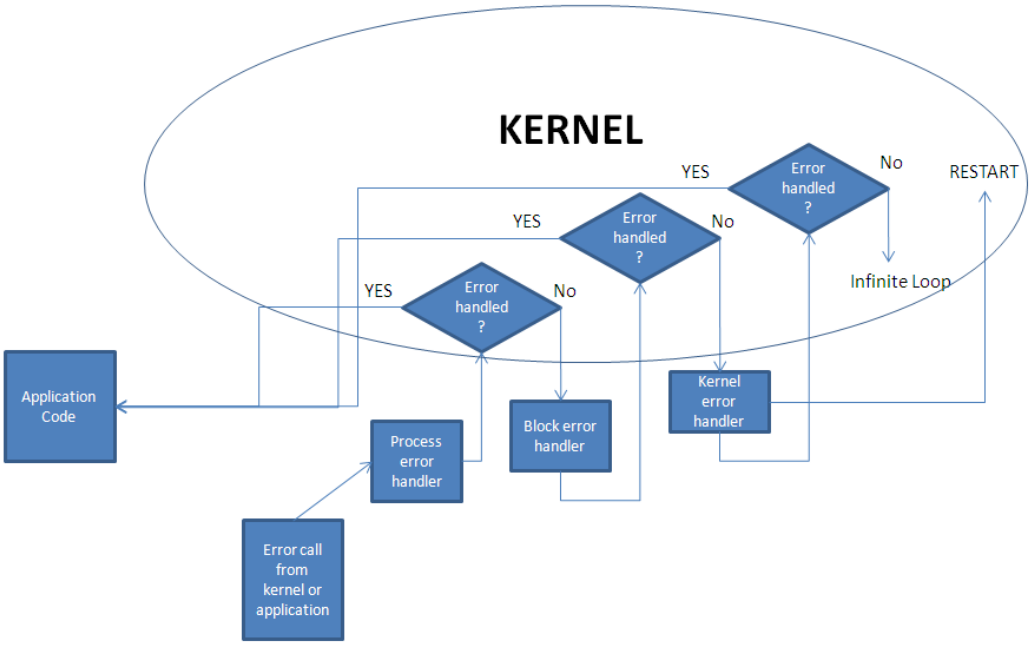


Figure 19: OSE error handling levels.

Error handlers in OSE are defined in the form of: PROC_ERROR_HANDLER (<procname>, <entrypoint>, <stack_size>) for processes and BLOCK_ERROR_HANDLER (<blockname>, <entrypoint>, <stack_size>) for blocks.

- Error codes: error codes in OSE are 32bit entities containing Fatal bit, Subcode and Error code bits patterns. The subcode refers to a subsystem; for example kernel uses (0x01000 - 0x530000)& OSE_ESUBCODE_MASK. Each subsystem has its own file describing possible errors.
- Memory protection: every accessible logical address in a memory-protected system must belong to a region. A region is either static or dynamic. Accessing an address outside a region leads to an access violation interrupt and eventually an OSE error. It is possible to define R/W/X (read/write/execute) permissions per region. This way OSE enables the following features: -Protection of code and read-only data against faulty write access. This is important for robustness of the system. -Protection of supervisor regions from user-mode accesses. -Protection of supervisor data from user write-accesses. -Inter-domain protection between application domains. This keeps faulty writes from causing interference between load modules.
- Several run-time configurations can be defined that are read and applied at startup time. Regarding error checks it is possible to define different error checking levels that are used to get information about incorrect usage of the operating system (using error check can slow down execution of system calls): 0. No error checking, 1. Only error checks with minor real-time impact are performed, 2. Full error checking. For example in the configuration, error checking level can be set as: krn/error_checks=2.
- Relaxing error check: This is another configuration (read at startup) to deactivate a number of error checks that have been added to OSE since the application was written. The value for this parameter is in the form of bit masks as follows:
  1: Disable the OSE_EILLEGAL_USER_MODE error, meaning that a supervisor process is trying to create a user mode block. This error is non-fatal and could instead when needed be acknowledged without action in an error handler. Even though it is not fatal, it is likely that user mode processes running in a supervisor program will cause memory protection exceptions.
  2: Permit a user mode process to create an interrupt or timer-interrupt process
  4: Permit a user mode process to create a supervisor process.
  8: Permit a user mode process to create a process in another memory domain.
  16: Permit the creation of child blocks when in shared mode.
  32: Permit user mode process to execute get_mem and set_mem in supervisor mode.
- Kernel Halt Handler: It is a static configuration and specifies the name of a user-provided kernel-halted handler. It can be seen as the last resort in the error handling; e.g. KERNEL_HALTED_HANDLER (bsp_reset).
- program handler, load module: restarting...

In summary, the mechanisms mentioned here directly or indirectly provide support for requirements **(Req 24)**, **(Req 25)**, **(Req 26)**, **(Req 27)**, **(Req 28)**, **(Req 29)**, **(Req 30)**.

3.4.5 Feasibility of Further Run-time Features Desired by End Users

(Req 34): *Per-group budgets:* It is possible make groups of task/processes and define memory budget on them. For monitoring, it is possible to implement that in a similar way as mentioned in the above item. But there is no support for specification of execution times. It can be implemented though in CHESS a prototype scheduler.

(Req 35) and **(Req 36)** are supported in OSE operating system and it provides APIs to get related values for these features. It is also possible to use OPTIMA profiling tool which has been designed for OSE and to monitor system behaviour. For instance as for CPU profiling in OSE, CPU usage can be measured in total for the whole target, per block or process, or per process priority. On OSE multicore systems, the execution unit (CPU core) on which to perform the profiling can also be selected (default is core 0) if not wanting to profile the whole system.

Requirements **(Req 37)** and **(Req 38)** are not supported but are implementable in OSE and **(Req 39)** can be partially supported (depending on the extent of run-time checks to disable; e.g. whether to disable even hardware-related run-time checks, etc.).

4 Conclusion

This document has collected requirements on run-time environments, in order to guarantee consistency between the CHESS analysis models developed in WP3 and WP4 and the CHESS run-time environments developed in WP5. The document serves as input for Task 5.3 on the implementation of run-time environments, and as input for Task 5.2 to ensure that platform-specific transformations target sufficiently predictable subsets of run-time environments.

We collected 39 requirements, 33 of which originated from the analysis models in WP3 and WP4. The other 6 requirements are end user requests for additional run-time features, some of which are specific to end user domains. While the requirements that stem from predictability analysis (WP4) are rather concise, the requirements from dependability analysis (WP3) are more general and will be further refined during use case development.

The requirements were evaluated and approved by the end users from the three targeted domains: telecommunications, space and railway. The most disputed issue was the requirement for a *static* set of threads, as imposed by schedulability analysis ((**Req 4**), (**Req 5**)). and failure propagation and transformation analysis ((**Req 31**)). While these requirements are in line with practise for the space and railway domain, they are too restrictive for the telecommunications domain, where dynamic allocation and disposal of threads and processes (as well as other resources) is common practise.

The introduction of this document posed the quest for a single multi-domain analysis model and, relatedly, for a multi-domain set of requirements on run-time environments. This document indicates that a common treatment for the space and railway domains seems entirely possible. Telecommunication systems differ in some respects from space and railway, for instance, in that dynamic resource management is used more extensively, and in that certain computation patterns that are common in space and railway (e.g., periodic tasks) are less common in telecommunications. While some domain-specific requirements may need to be addressed separately, we remain hopeful that a uniform treatment for all three domains will be feasible to a large extent.

The feasibility of the requirements has been evaluated for the run-time environments Ada Real-time, Java/RTSJ, Real-time Linux and OSE, and it has been determined in how far the requirements are already supported and in what ways the existing run-time environments need to be extended. The requirements in support of the analysis methods from WP4 are feasible on all run-time environments, in some cases after implementing appropriate extensions and with the help of model-to-code transformers omitting run-time features that hamper analysability. As mentioned above, the requirements in support of dependability (WP3) are of a more general nature. Consequently, it was hard to judge their feasibility with yes/no answers. We addressed these requirements by identifying supporting

mechanisms that the various run-time environments offer. We expect that more concise dependability requirements will evolve from the use cases. Run-time environments will be adapted to support such evolving needs throughout the remainder of the project.

References

[1] Ada 2005 reference manual - language and standard libraries, international standard iso/iec 8652/1995 (e), with technical corrigendum 1 and amendment 1, iso/iec 8652:2007(e) ed. 3. `http://www.adaic.org/resources/add_content/standards/05rm/RM-Final.pdf`.

[2] *ObjectAda Real-Time V7.2 SPARC x SPARC/RAVEN Cross Development Guide*.

[3] Real-time specification for Java, version 1.0.2. `http://www.rtsj.org/specjavadoc/book_index.html`, 2003.

[4] JSR 282: Real-time specification for Java, version 1.1, early draft review. `http://jcp.org/en/jsr/detail?id=282`, 2009.

[5] *JamaicaVM 6.0 User Manual*, July 2010. `http://www.aicas.com/jamaica/6.0/doc/jamaicavm_manual.pdf`.

[6] Li Gong and Gary Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2nd edition, 2003.

[7] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

A Appendix

A.1 Requirements for Failure Mode and Effect Analysis

This appendix is a supplement to Section 2.3. It collects requirements on the System C virtual prototyping platform that is used for failure modes and effect analysis (FMEA). These requirements have not been included in Section 2.3, because they are only needed for the System C virtual prototyping platform, but not to the other target runtime environments.

Requirements for the FMEA analysis:

1. The platform must be configurable for analysis of different scenarios.
2. Different errors must be injected during execution. Type, location and time must be configurable.
3. A monitoring and checking strategy for failure detection and classification is required.
4. For statistically based analysis the simulation execution must be fully automatic.
5. For error injection a scenario definition which provides parameters ranges such as simulation time, is needed.
6. For dependability modelling the analysis must provide failure classifications and links between them.